Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 9
2024-4-22

Outline: Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Set cover
- Take-home messages

Outline: Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Set cover
- Take-home messages

Slides courtesy:
https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf

❖ Optimization Problems:

Given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

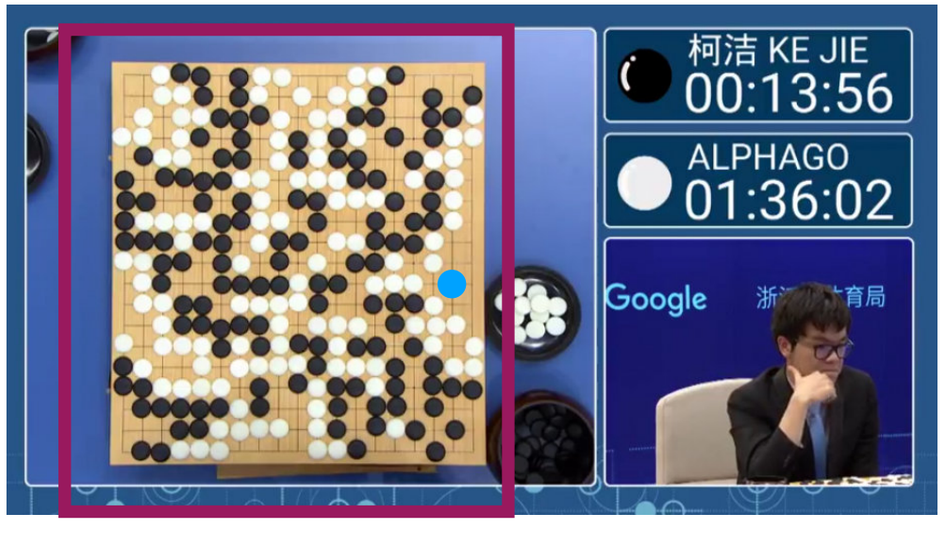
❖ Optimization Problems:

Given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

❖ The Greedy Method:

Make the best decision at each stage, under some greedy criterion. A decision made in one stage is not changed in a later stage, so each decision should assure feasibility.

The Decision Problem



- The agent faces with a series of "states".
- Need to choose the corresponding "actions".
- Each action has a reward.
- Target: maximize the total reward in a decision sequence by always choosing the right action.

Sometimes, we can be myopic (greedy) while still achieving the global optimal.

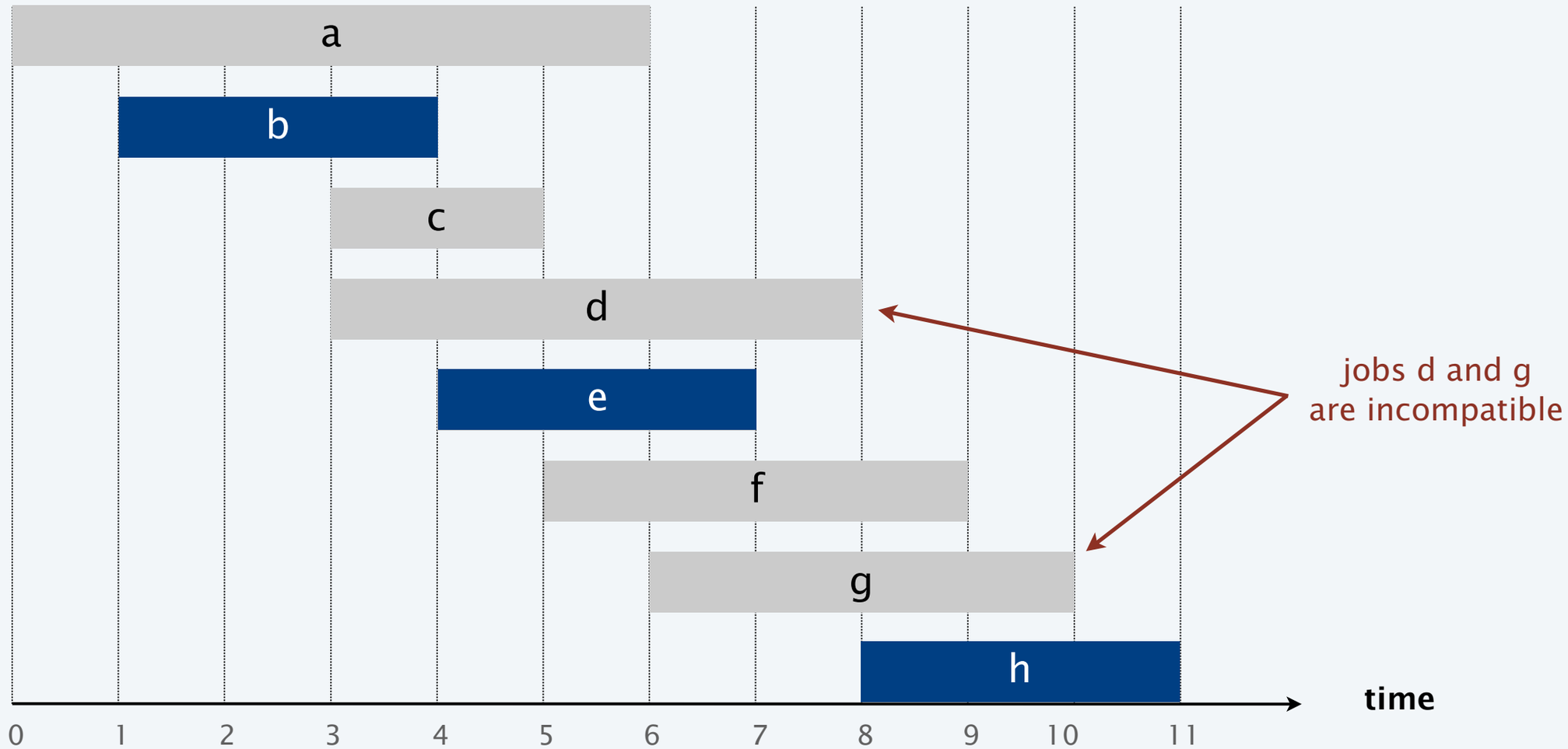
Greedy Algorithms

- Many classical algorithms not discussed in this lecture:
  - Minimum spanning trees
  - Dijkstra's shortest path algorithm
  - …

Similar to dynamic programming, it is beneficial to do more exercises.

Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Greedy algorithms I: quiz 2

Consider jobs in some order, taking each job provided it's compatible with the ones already taken. Which rule is optimal?

A. [Earliest start time] Consider jobs in ascending order of s_j.

B. [Earliest finish time] Consider jobs in ascending order of f_j.

C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. None of the above.

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

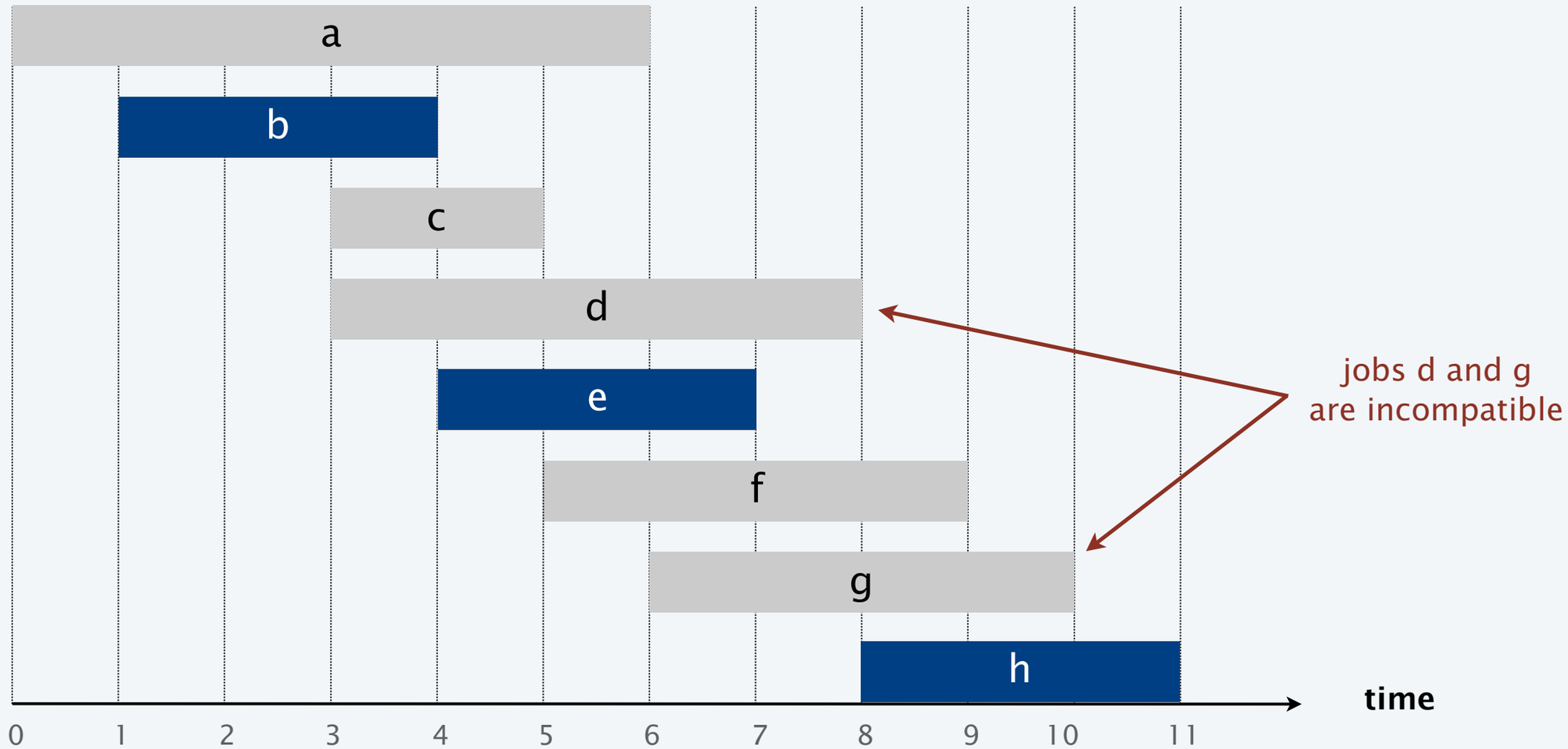
RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- Keep track of job j^* that was added last to S.
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

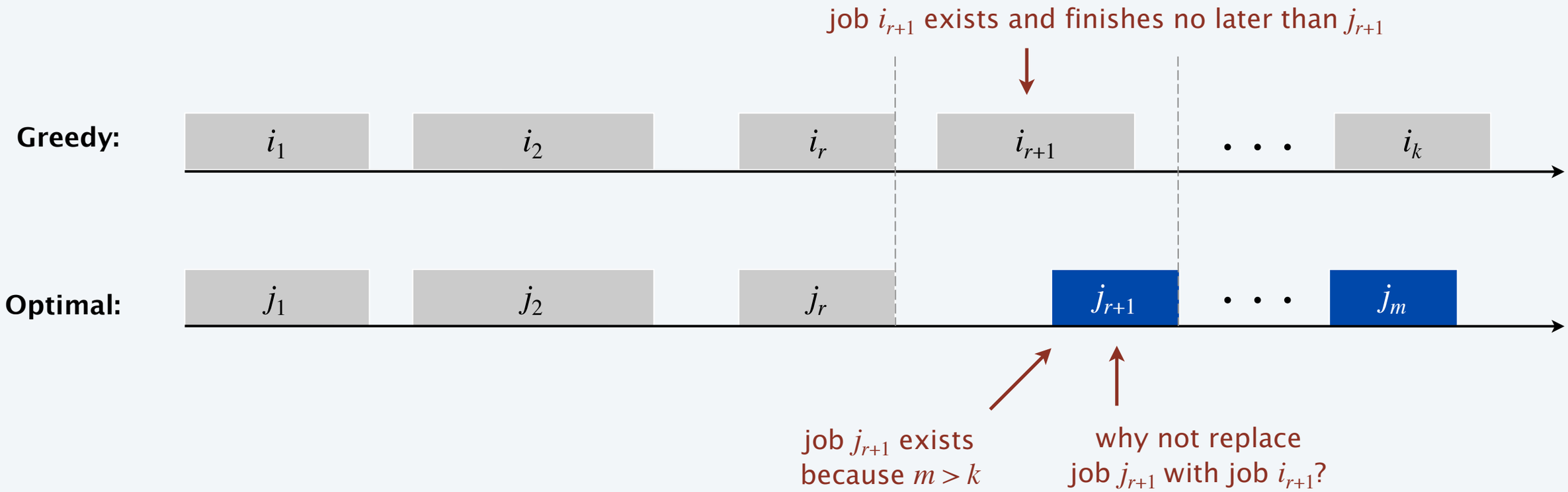


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

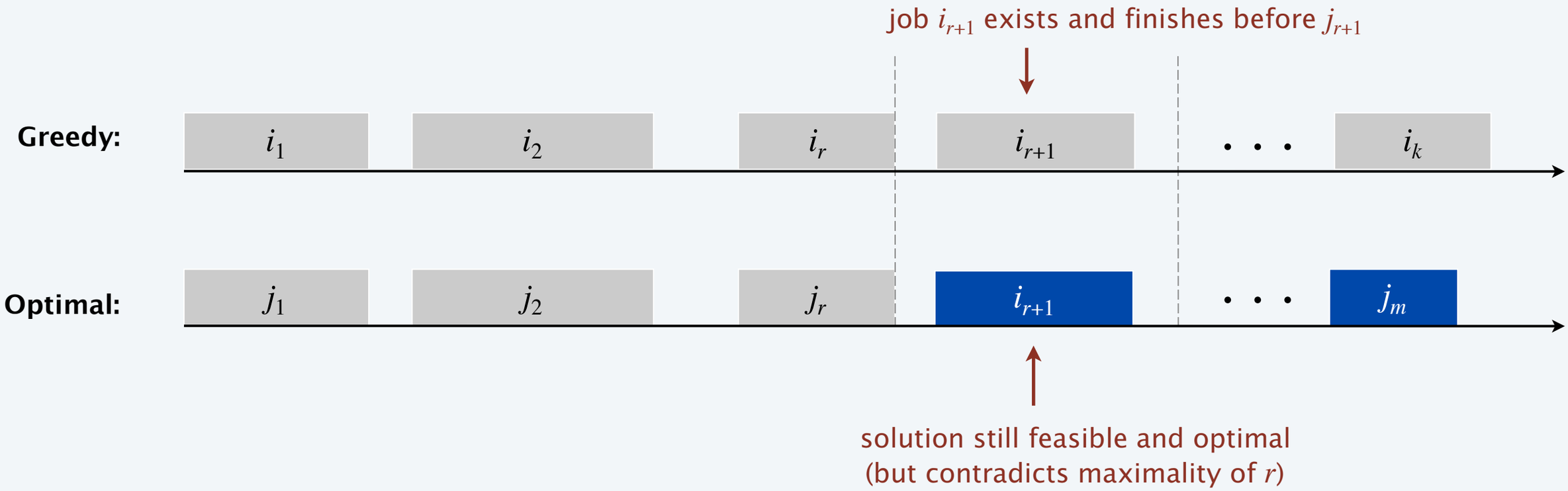


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Greedy algorithms I: quiz 3

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is the earliest-finish-time-first algorithm still optimal?

A. Yes, because greedy algorithms are always optimal.

B. Yes, because the same proof of correctness is valid.

C. No, because the same proof of correctness is no longer valid.

D. No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.

☞ Another Look at DP Solution

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + w_j \} & \text{if } j > 1 \end{cases}$$

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + w_j \} & \text{if } j > 1 \end{cases}$$

Q1: Is the DP solution still correct?
Q2: Is the Greedy solution still correct?

Elements of the Greedy Strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution

Outline: Greedy Algorithms

- Active selection
- Huffman coding
- Set cover
- Take-home messages

Huffman Codes – for file compression

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take ? bits to store the string as 1000 one-byte characters.

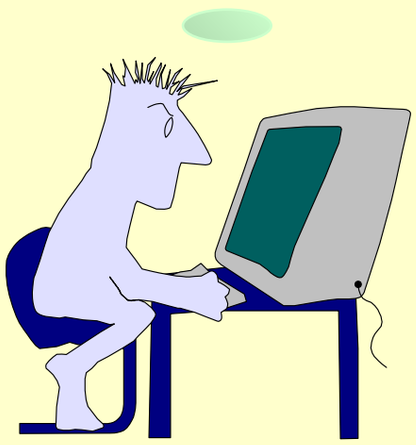
Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

Huffman Codes – for file compression

〖Example〗 **Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.**

Notice that we have only
4 distinct characters in that string.
Hence we need only
2 bits to identify them.



Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where C is the size of the character set */

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ **frequency ::=** number of occurrences of a symbol.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ **frequency ::=** number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ frequency ::= number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ **frequency ::=** number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111. ➡ 000101100010111

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ frequency ::= number of occurrences of a symbol.

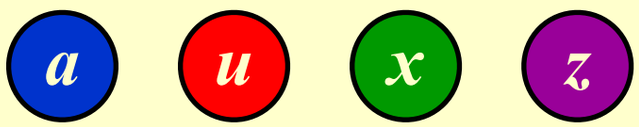
In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111. ➡ 00010110010111

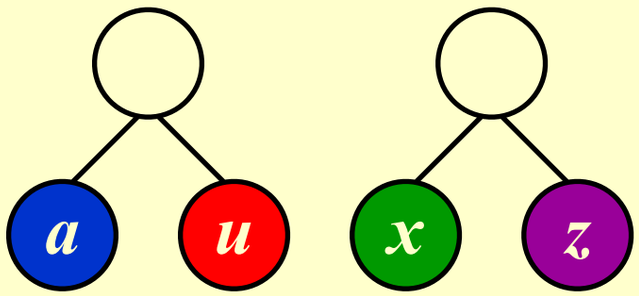
Note: If all the characters occur with the same frequency, then there are not likely to be any savings.

Representation of the original code in a binary tree /* trie */

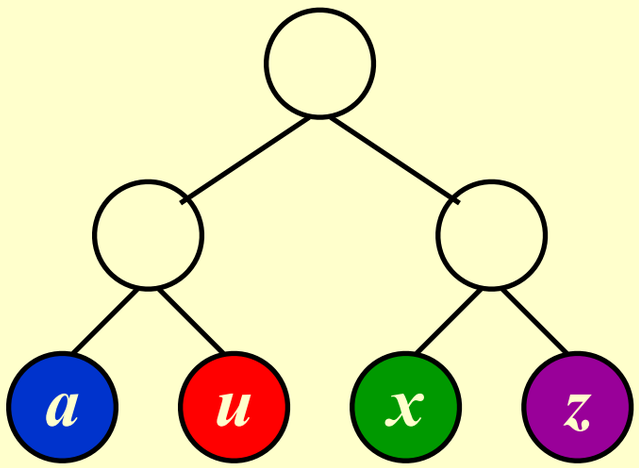
Representation of the original code in a binary tree /* trie */



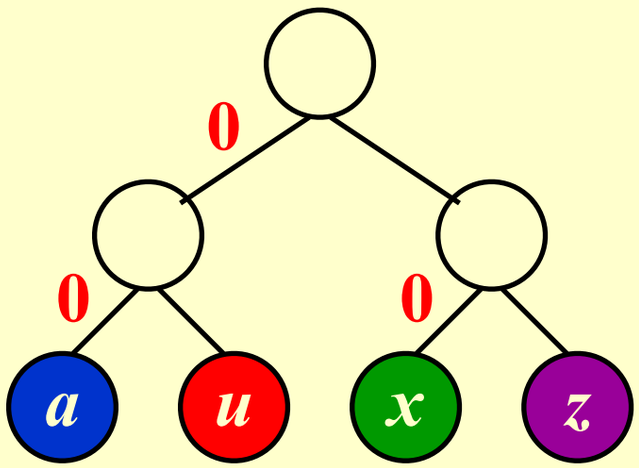
Representation of the original code in a binary tree /* trie */



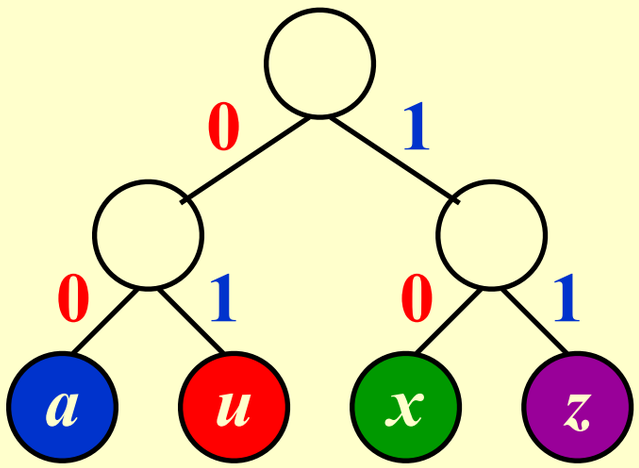
Representation of the original code in a binary tree /* trie */



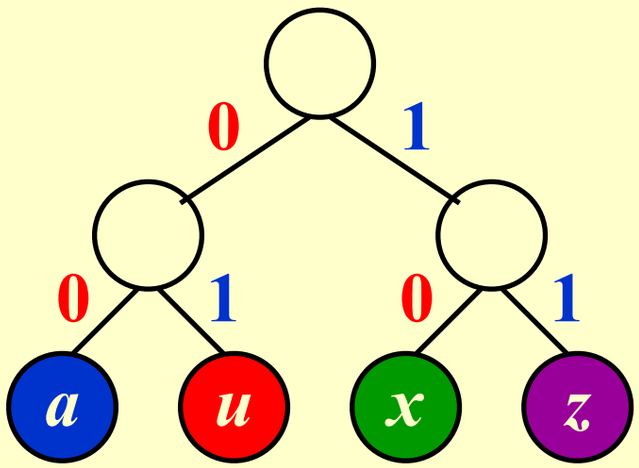
Representation of the original code in a binary tree /* trie */



Representation of the original code in a binary tree /* trie */

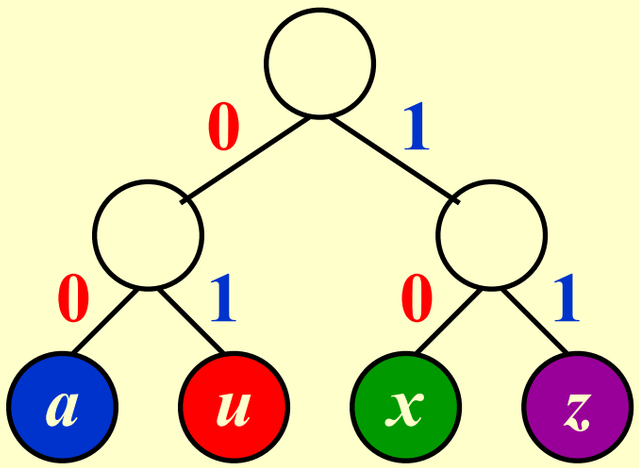


Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

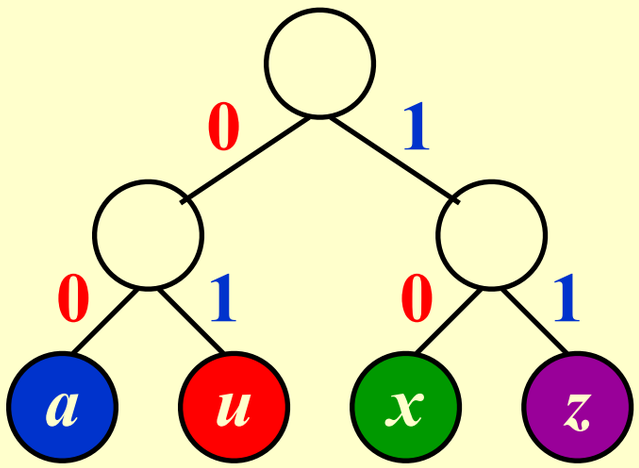
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the original code in a binary tree /* trie */

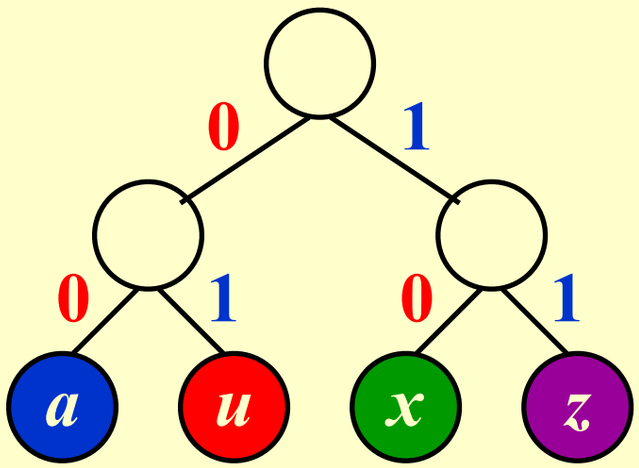


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

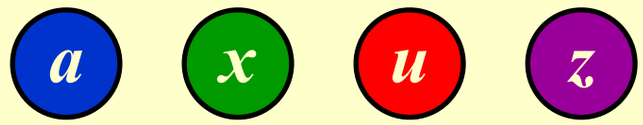
Representation of the original code in a binary tree /* trie */



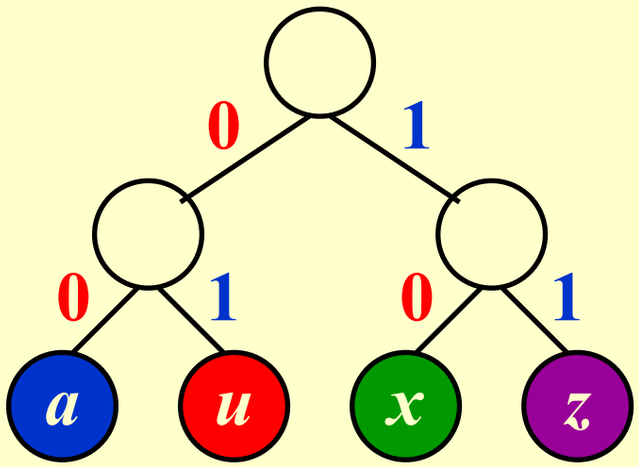
➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



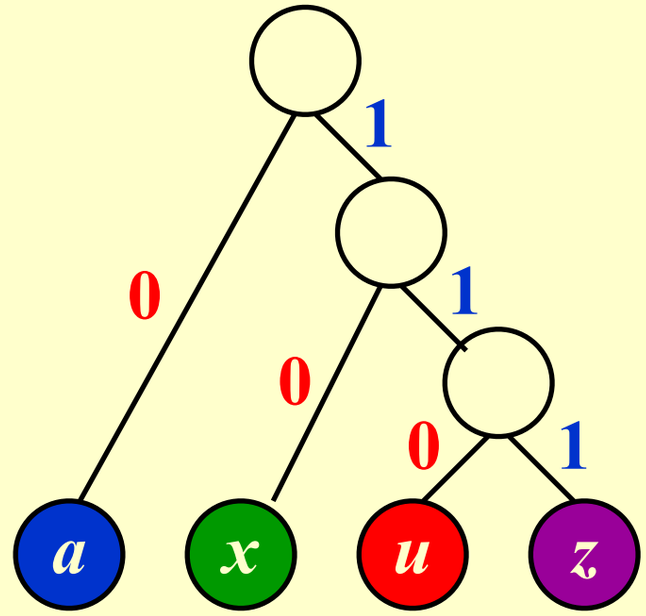
Representation of the original code in a binary tree /* trie */



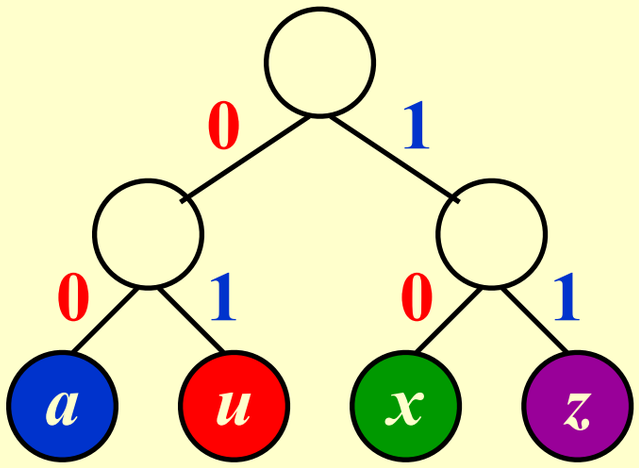
➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



Representation of the original code in a binary tree /* trie */

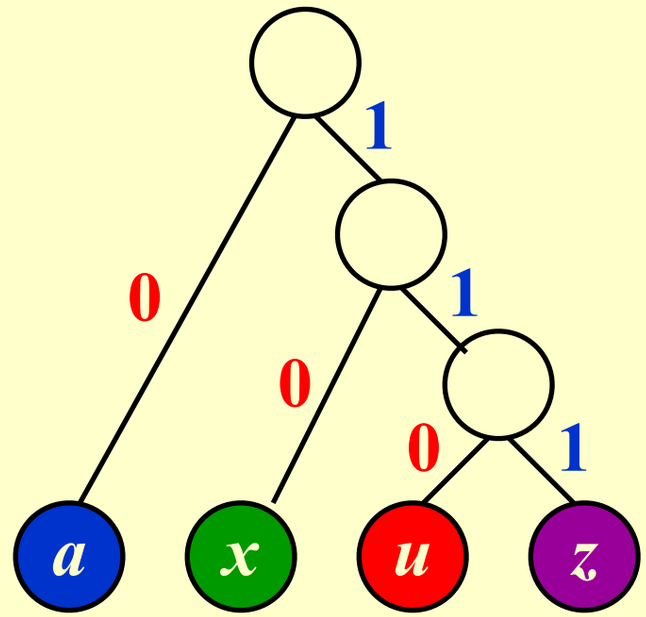


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

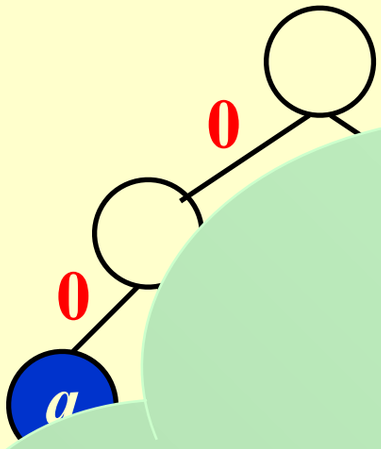
Cost **(*aaaxuaxz* → 000000100100101 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost **(*aaaxuaxz* → 000101100010111)**
= 1×4 + 3×1 + 2×2 + 3×1 = 14



Representation of the original code in a binary tree /* trie */



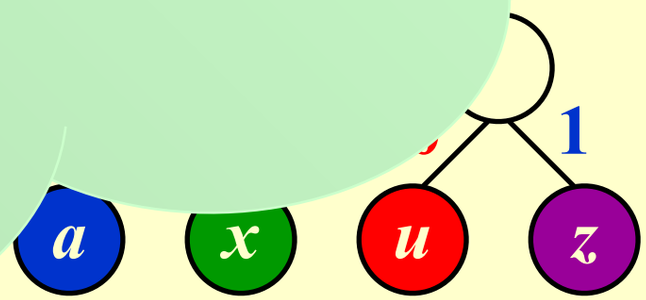
➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code is $\sum_i d_i f_i$.**

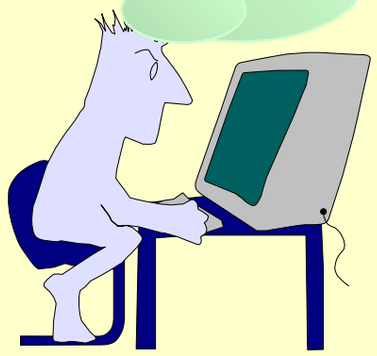
001011)

Now, with

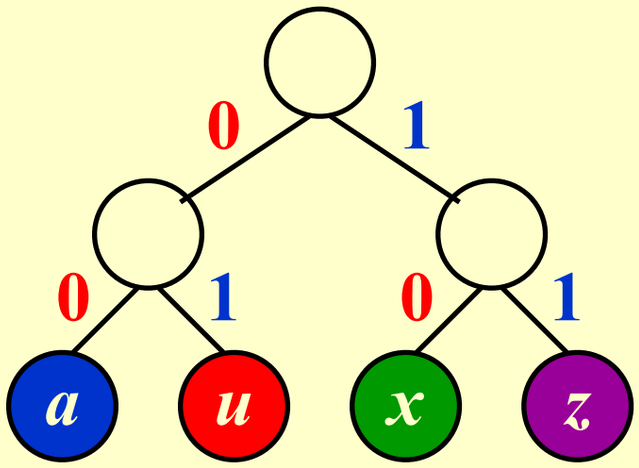
$a = 0$, $u = 110$, $x = 10$, $z = 111$

and the string 000101100101111, can you decode it?





Representation of the original code in a binary tree /* trie */

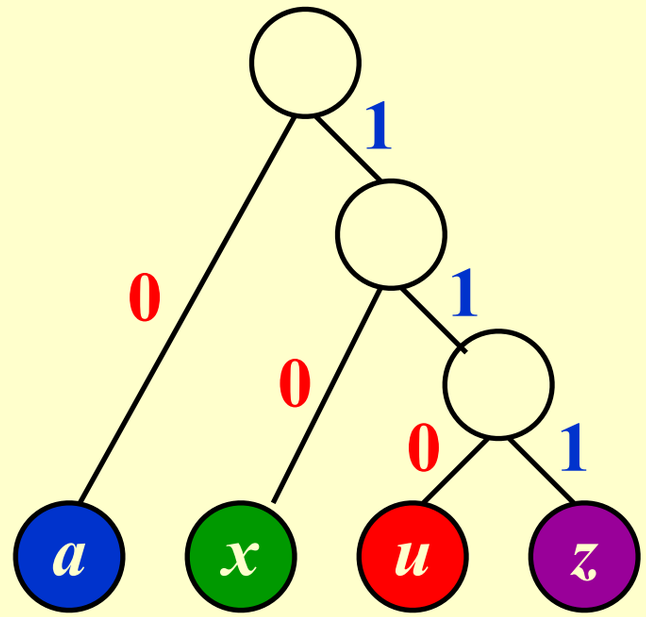


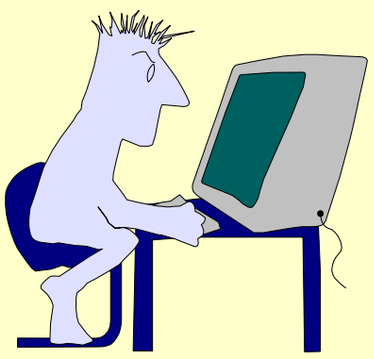
➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

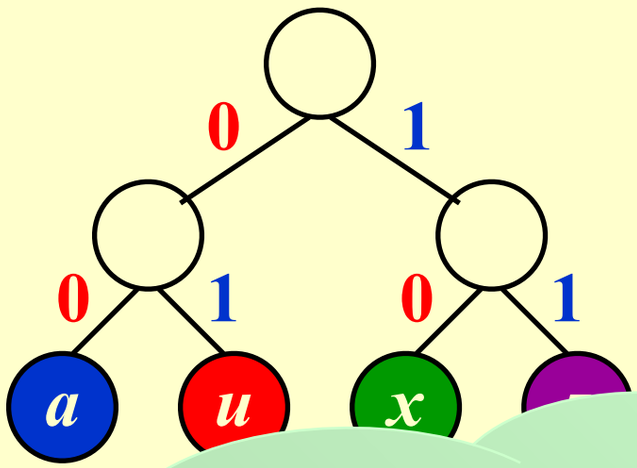
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree





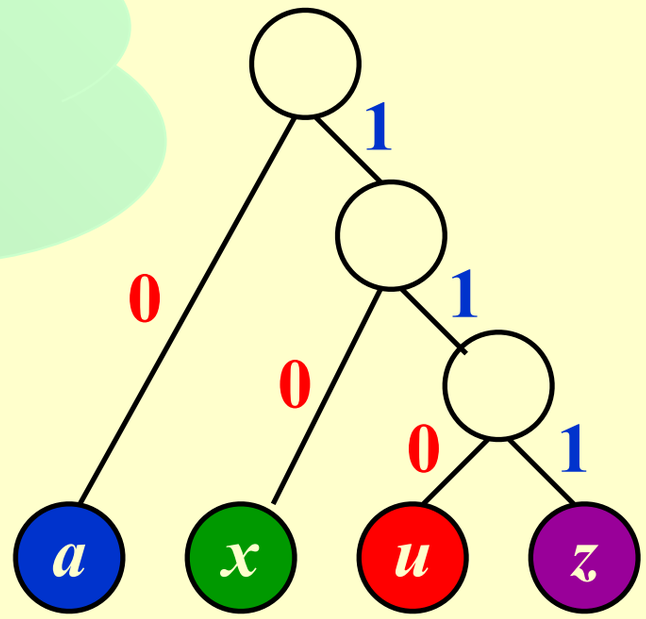
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

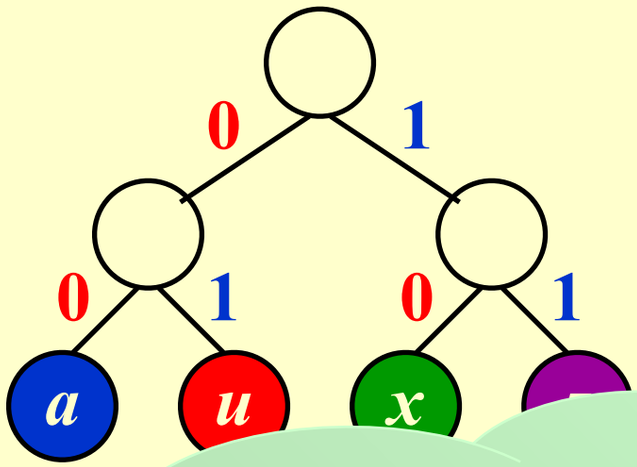
Cost **(*aaaxuaxz* → 00000010010011 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



The answer is *aaaxuaxz* (with *a* = 0, *u* = 110, *x* = 10, *z* = 111). What makes this decoding method work?

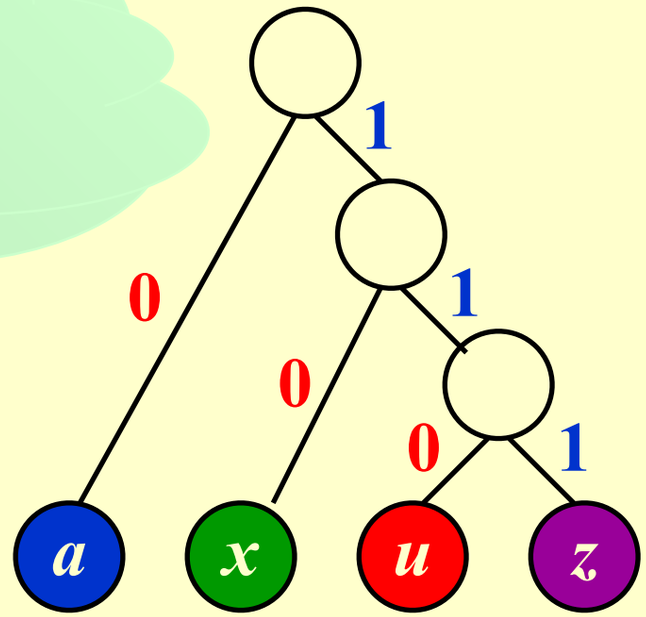
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

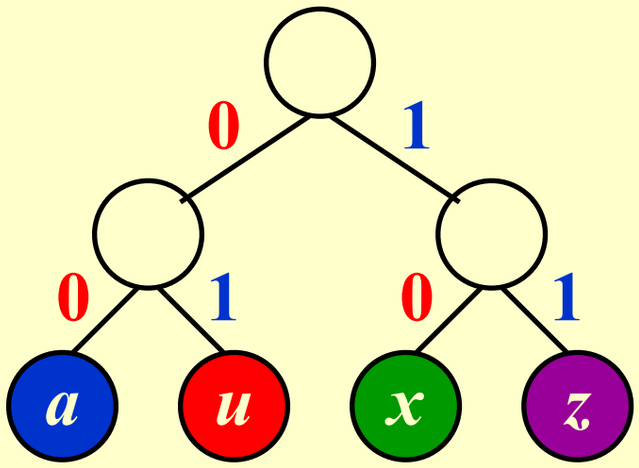
Cost **(*aaaxuaxz* → 00000010010011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



The trick is:
No code is a *prefix* of another.

Representation of the original code in a binary tree /* trie */

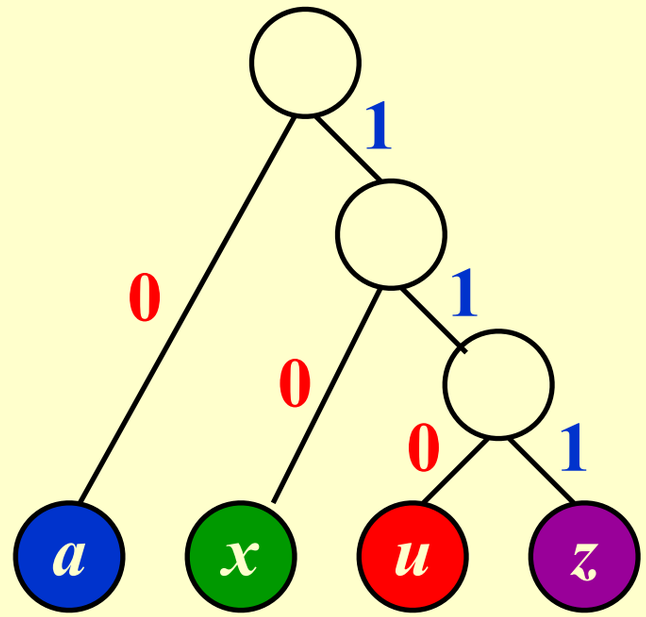


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

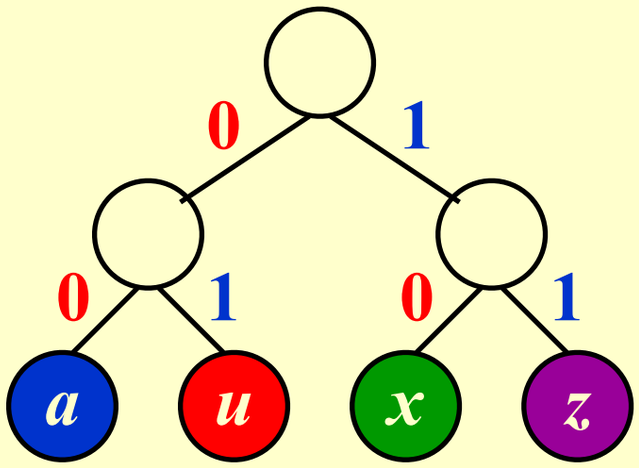
Cost **(*aaaxuaxz* → 000000100100101 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost **(*aaaxuaxz* → 000101100010111)**
= 1×4 + 3×1 + 2×2 + 3×1 = 14



Representation of the original code in a binary tree /* trie */

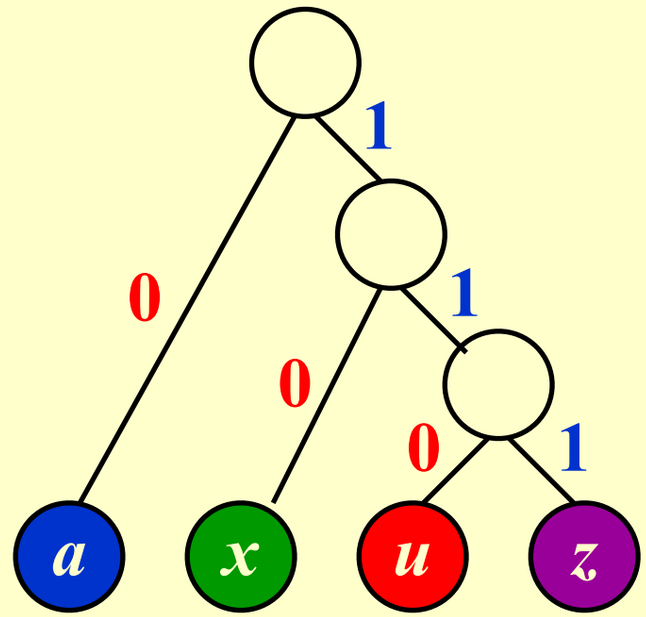


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

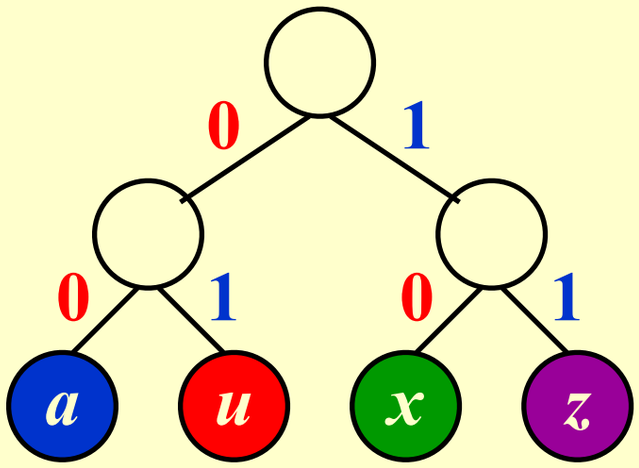
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree



Discussion 13: **What must the tree look like if we are to decode unambiguously?**

Representation of the original code in a binary tree /* trie */

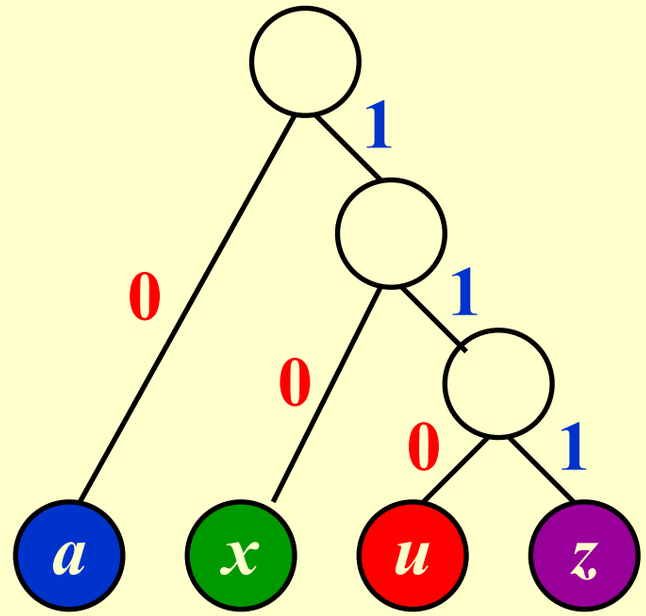


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

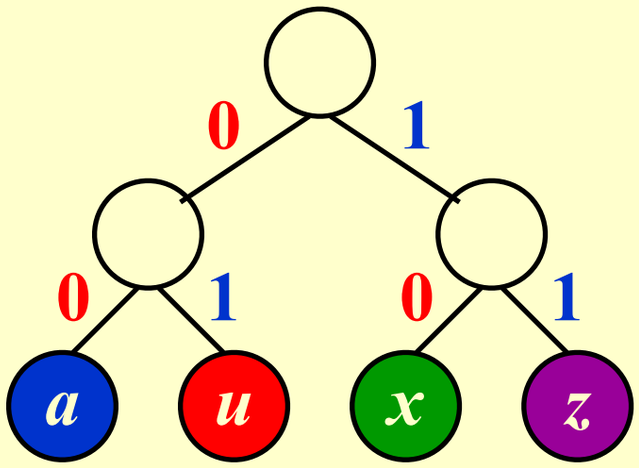
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree



Any sequence of bits can always be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Representation of the original code in a binary tree /* trie */

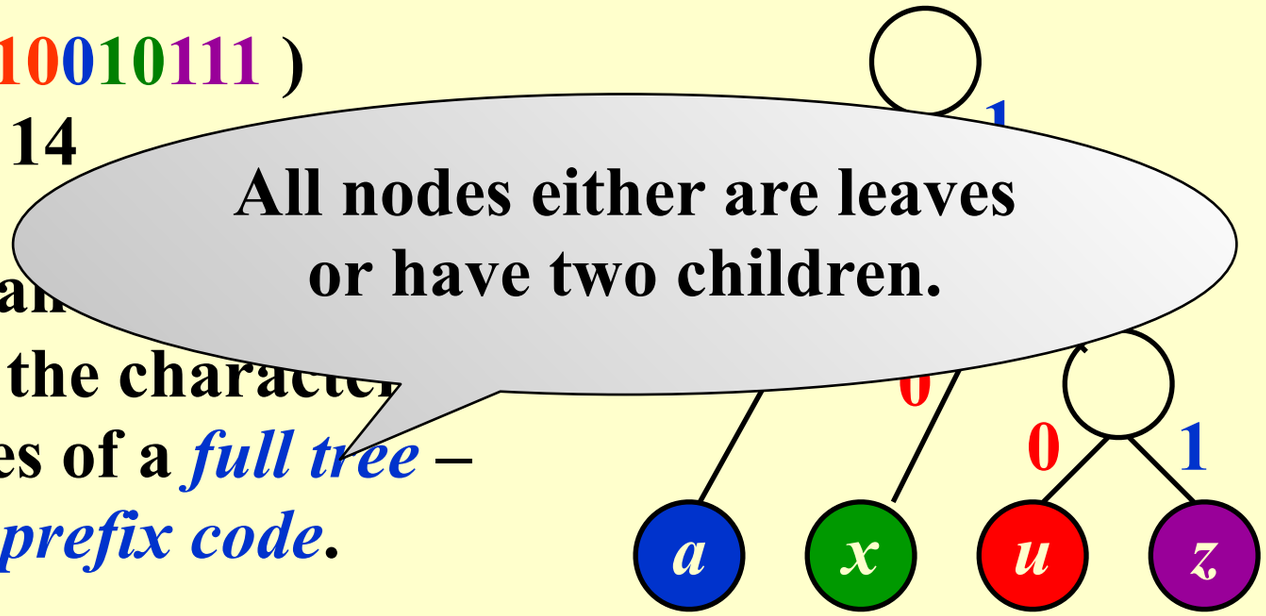


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

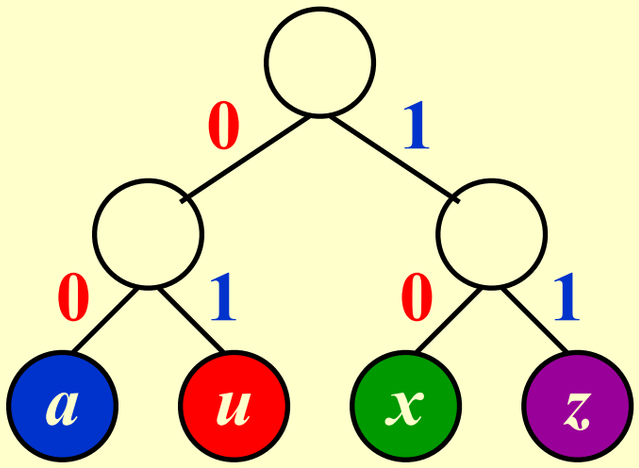
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree



Any sequence of bits can be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Representation of the original code in a binary tree /* trie */

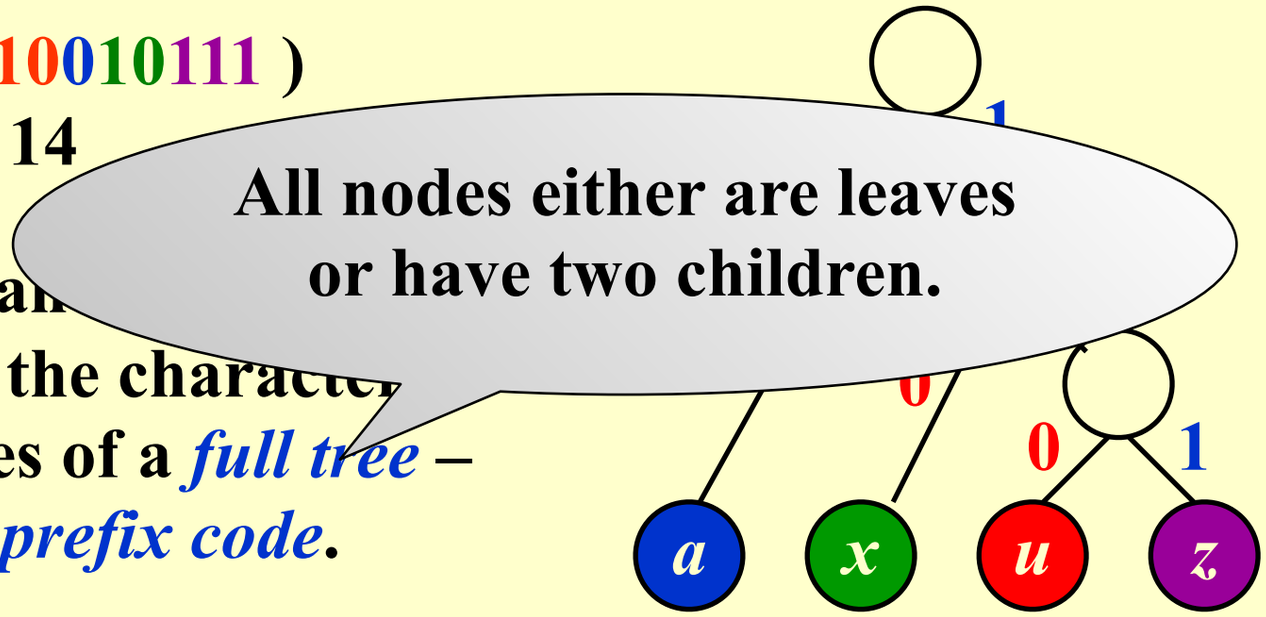


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14



Any sequence of bits can be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Find the full binary tree of minimum total cost where all characters are contained in the leaves.

➢ Huffman's Algorithm (1952)

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
    }
}
```

$$T = \mathrm{O}(\quad ? \quad)$$

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

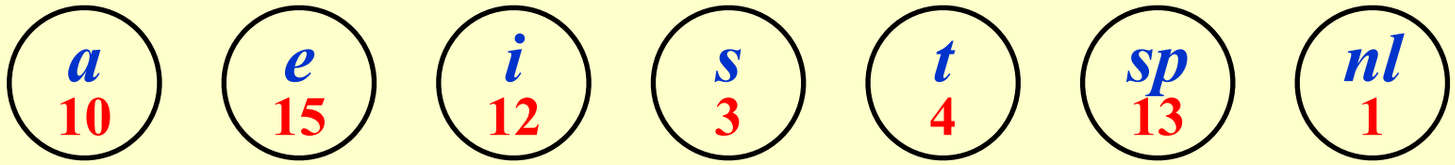
$$T = \mathrm{O}(\quad C \log C \quad)$$

〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1

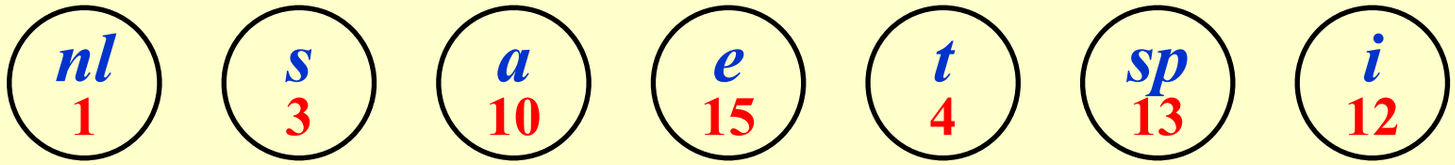
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



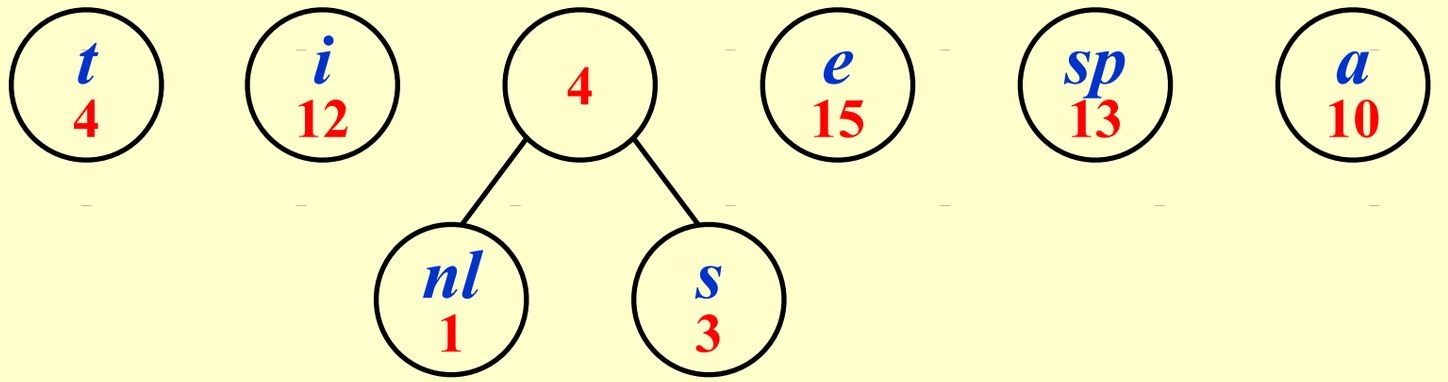
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



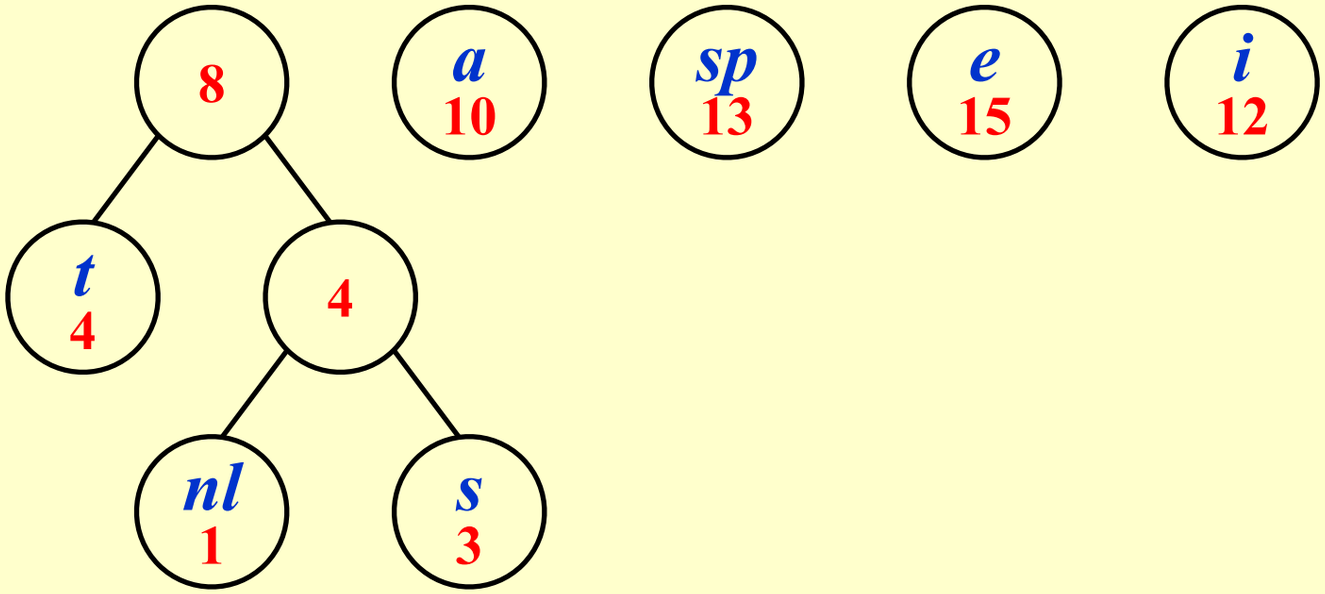
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



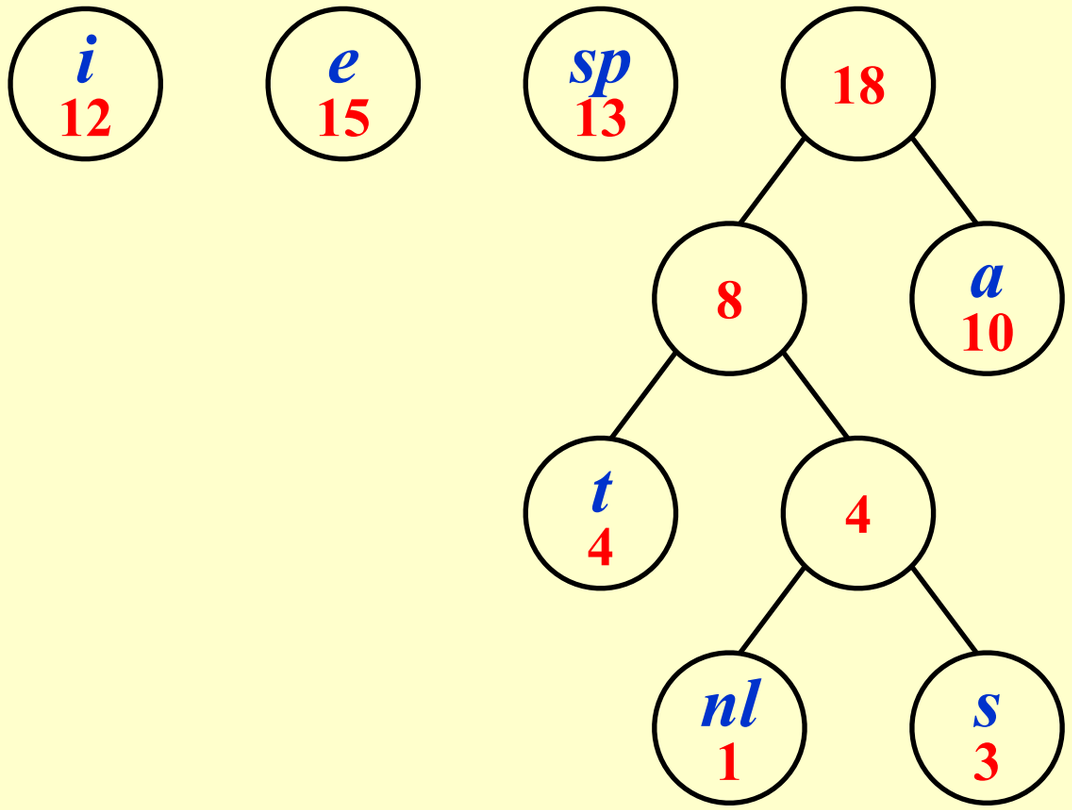
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



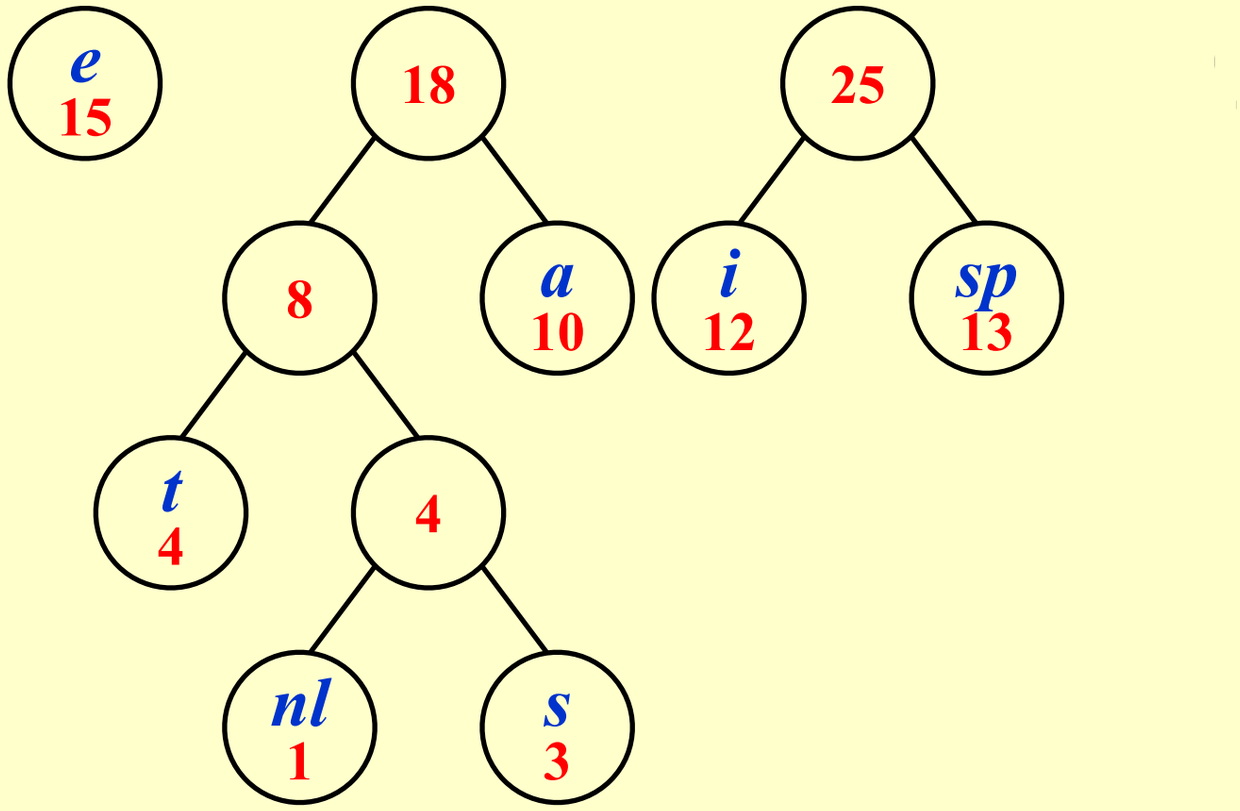
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



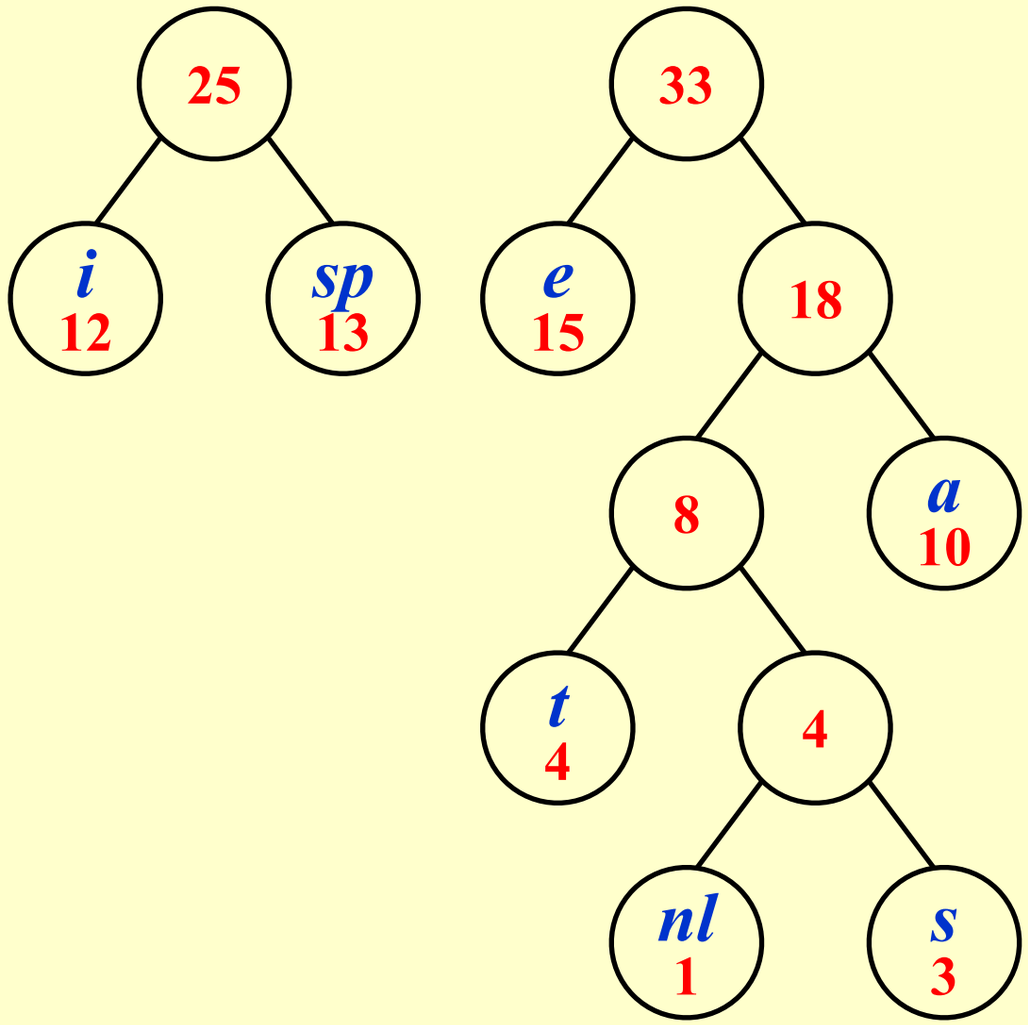
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



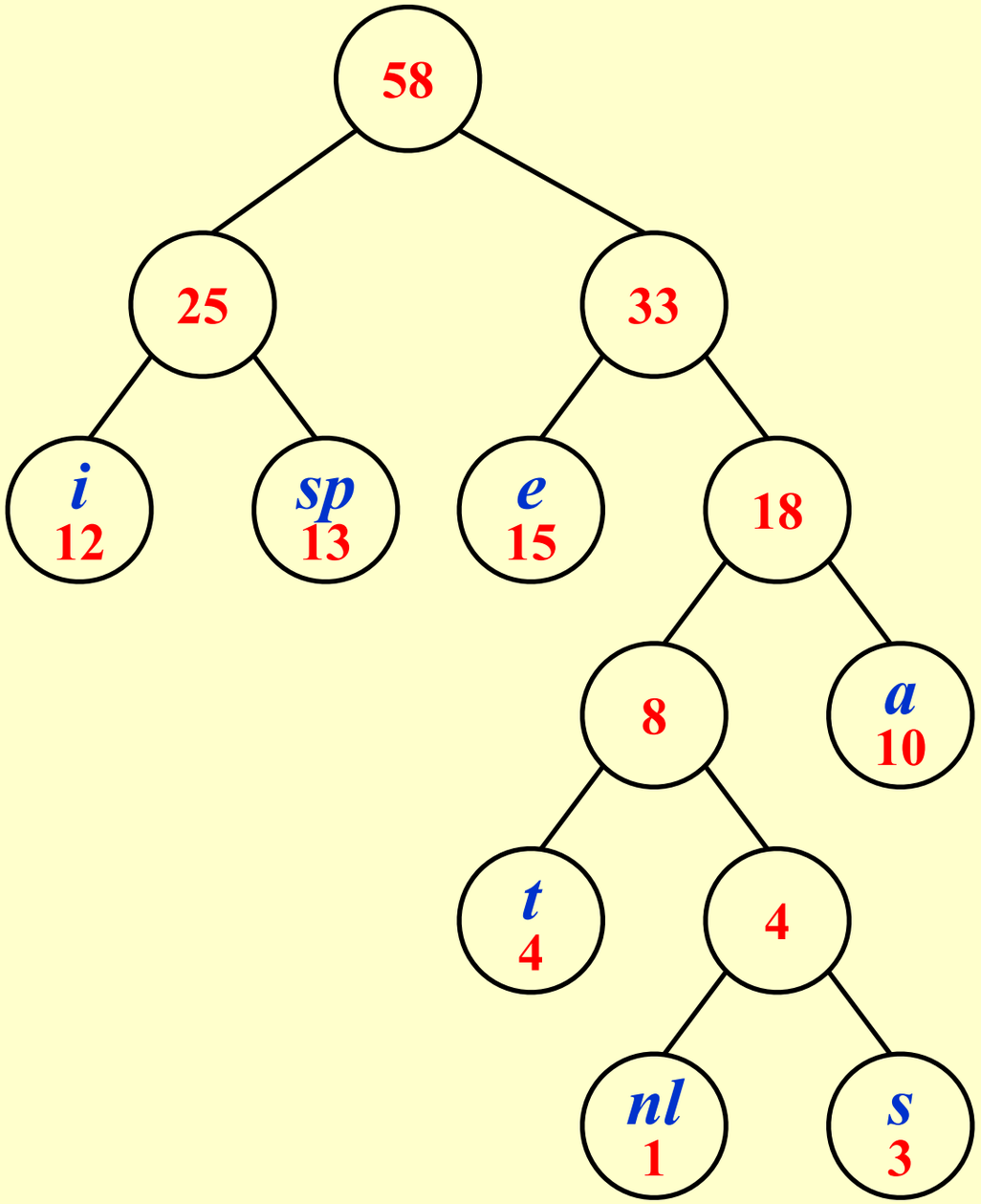
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



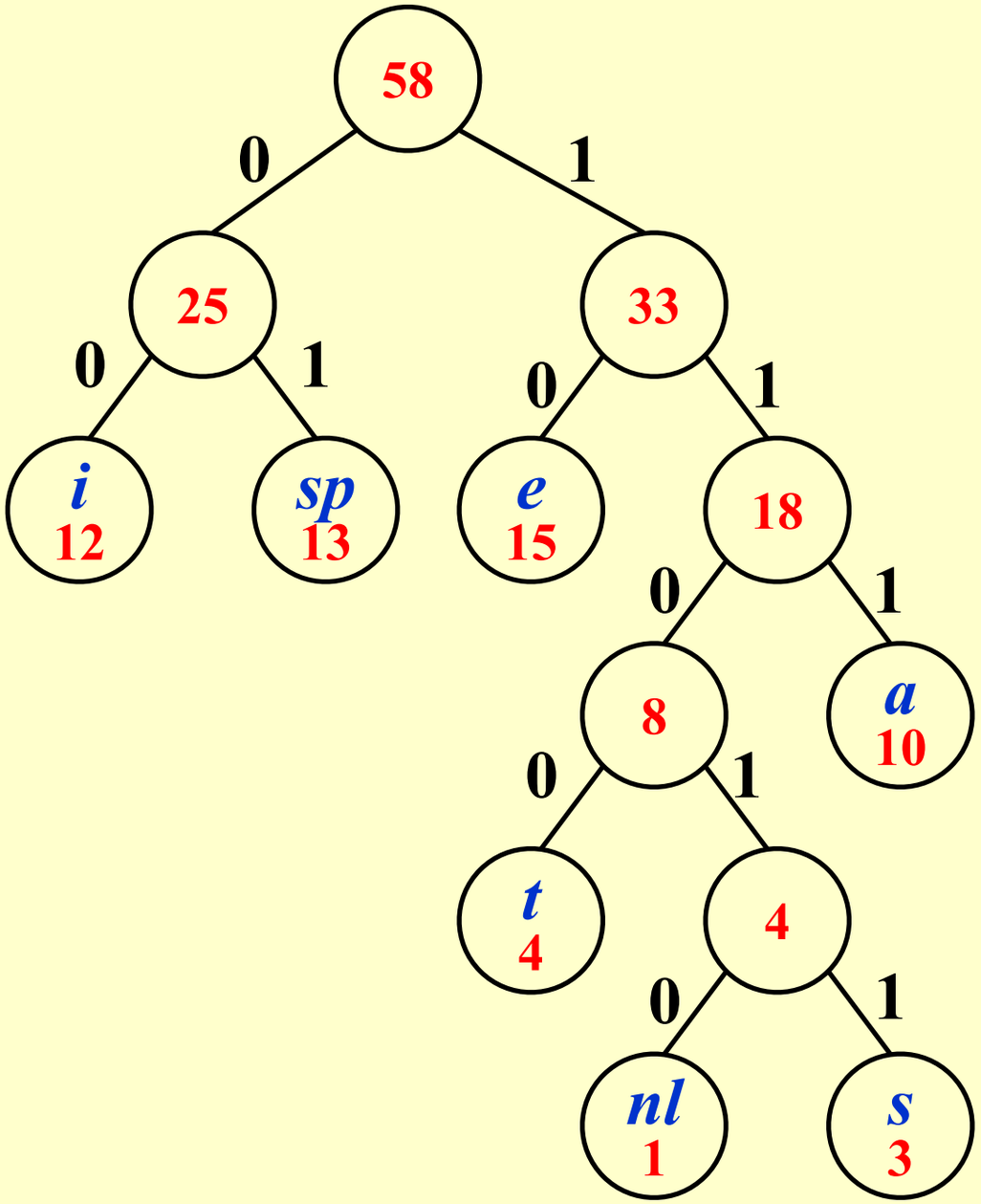
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



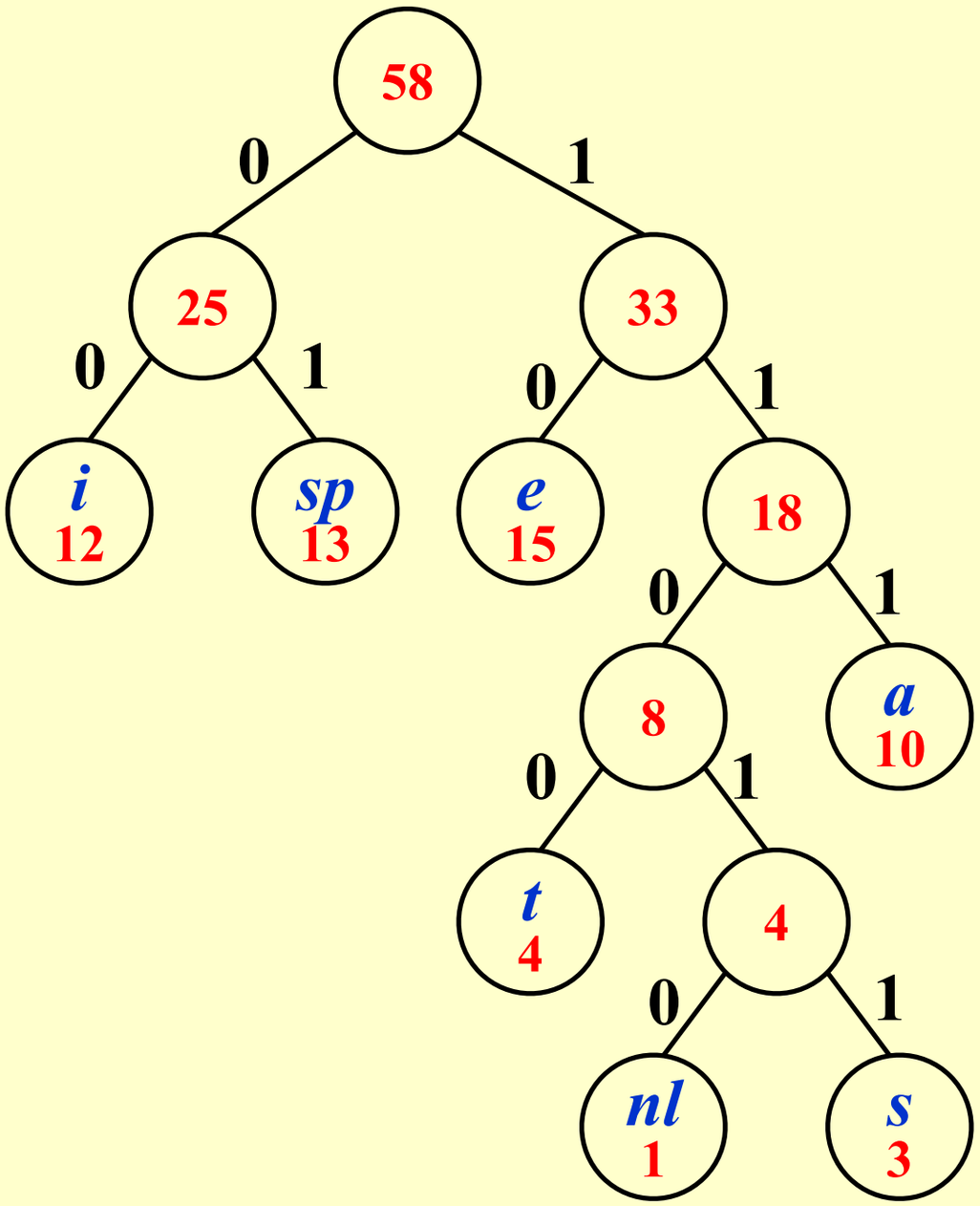
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



〖Example〗

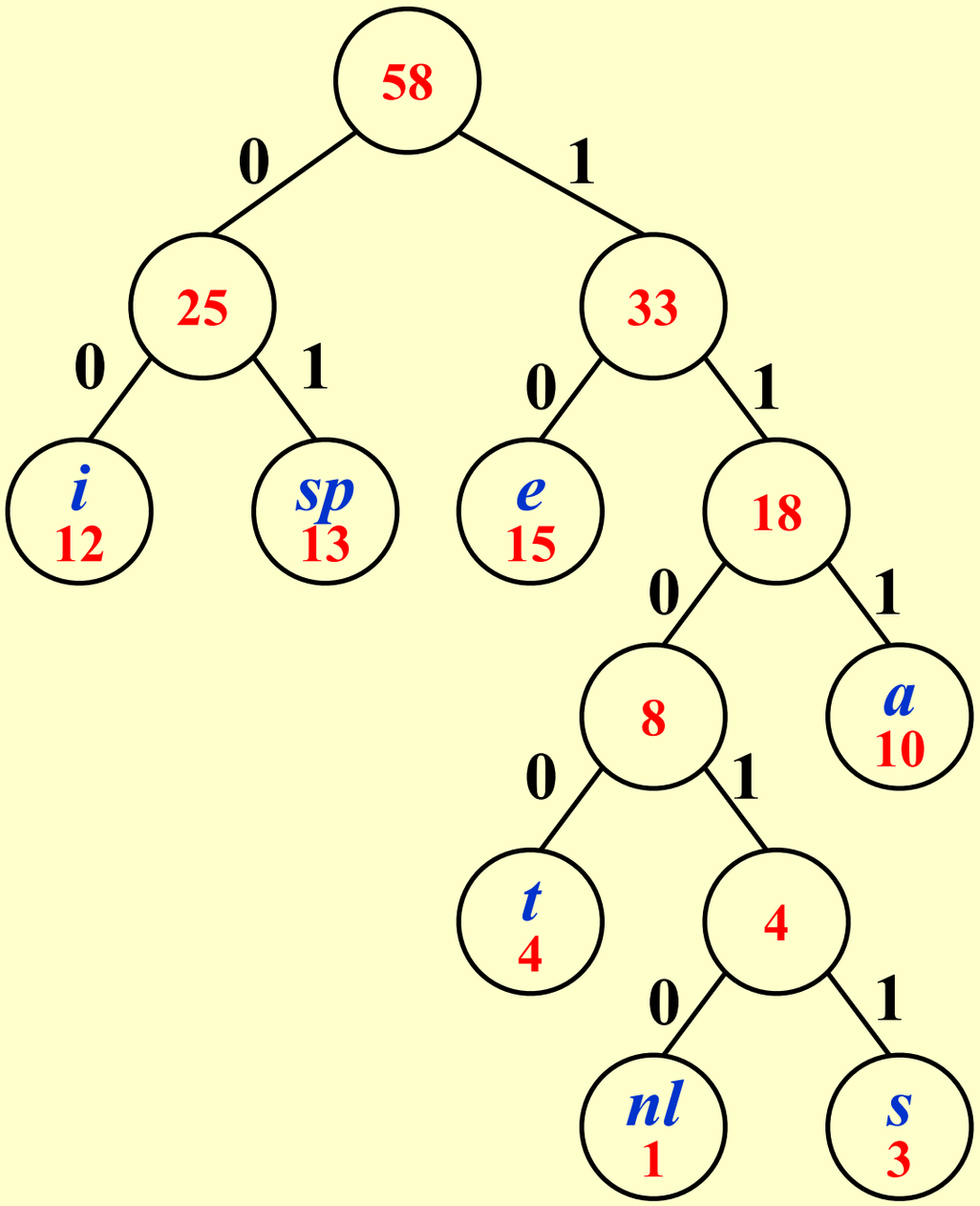
C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

$$Cost = 3\times10 + 2\times15 + 2\times12 + 5\times3 + 4\times4 + 2\times13 + 5\times1 = 146$$

Correctness:

① The greedy-choice property

Correctness:

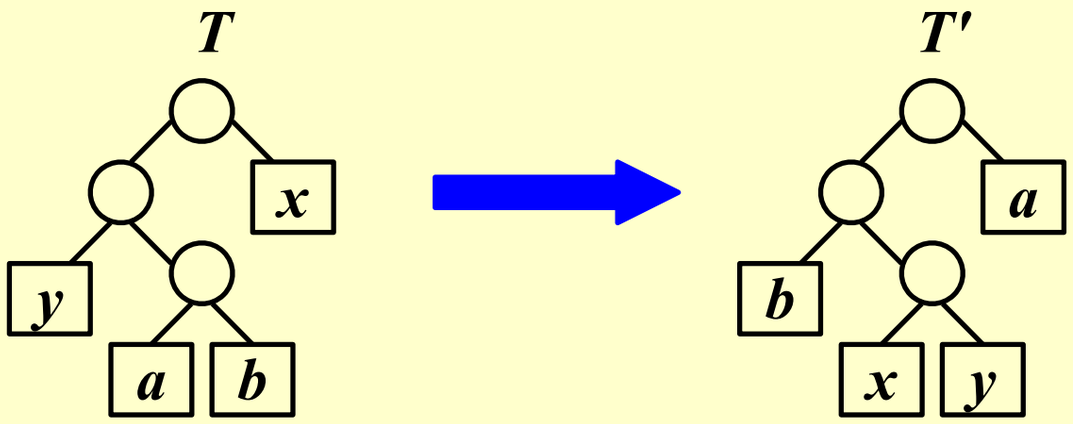
① The greedy-choice property

【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Correctness:

① The greedy-choice property

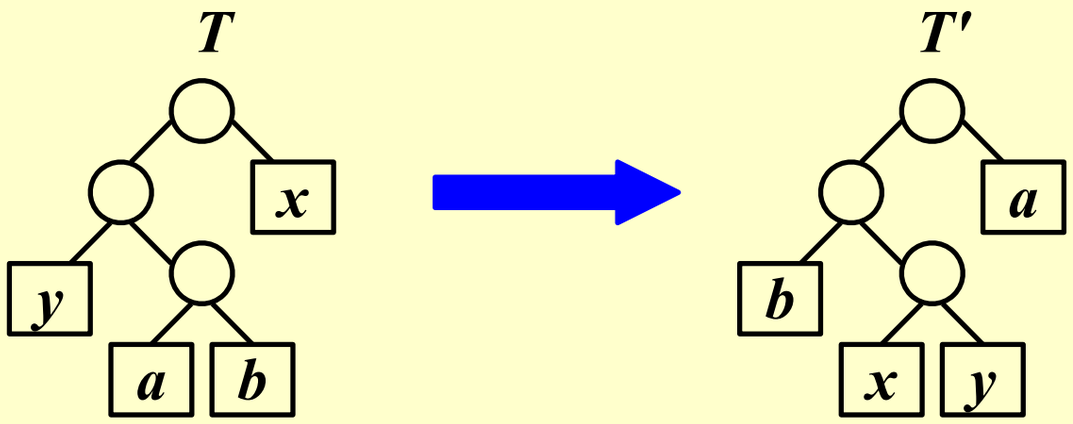
【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Correctness:

① The greedy-choice property

【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



$$\text{Cost}(T') \le \text{Cost}(T)$$

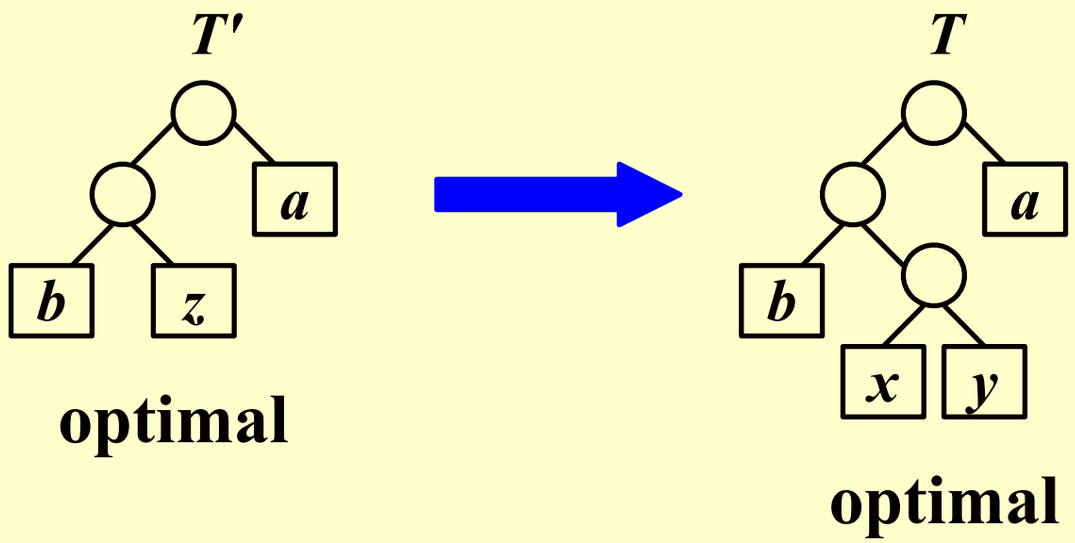
② The optimal substructure property

② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

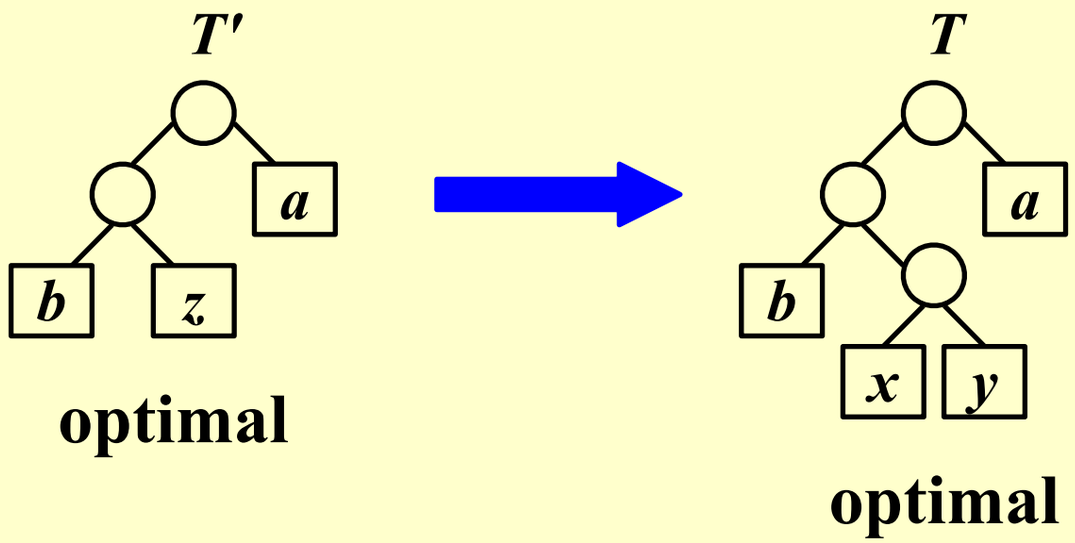
② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.



② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

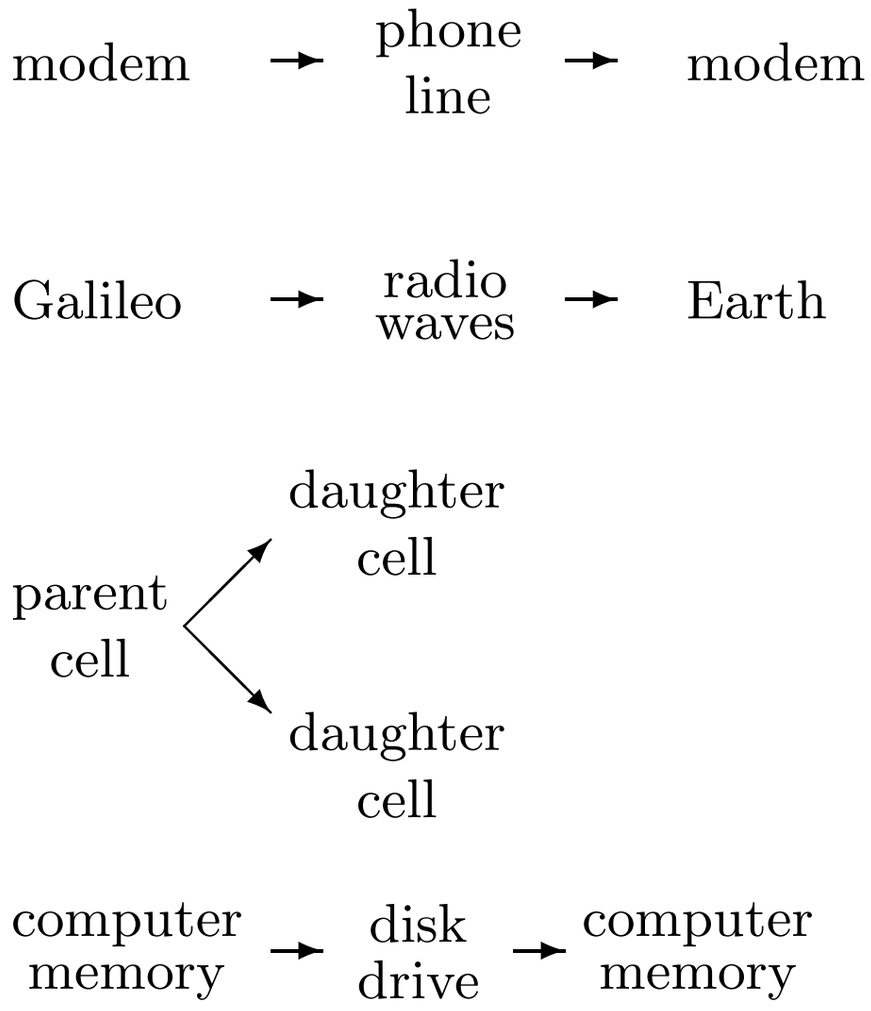


By contradiction.

What is Information?

She says hello to me, she loves me?

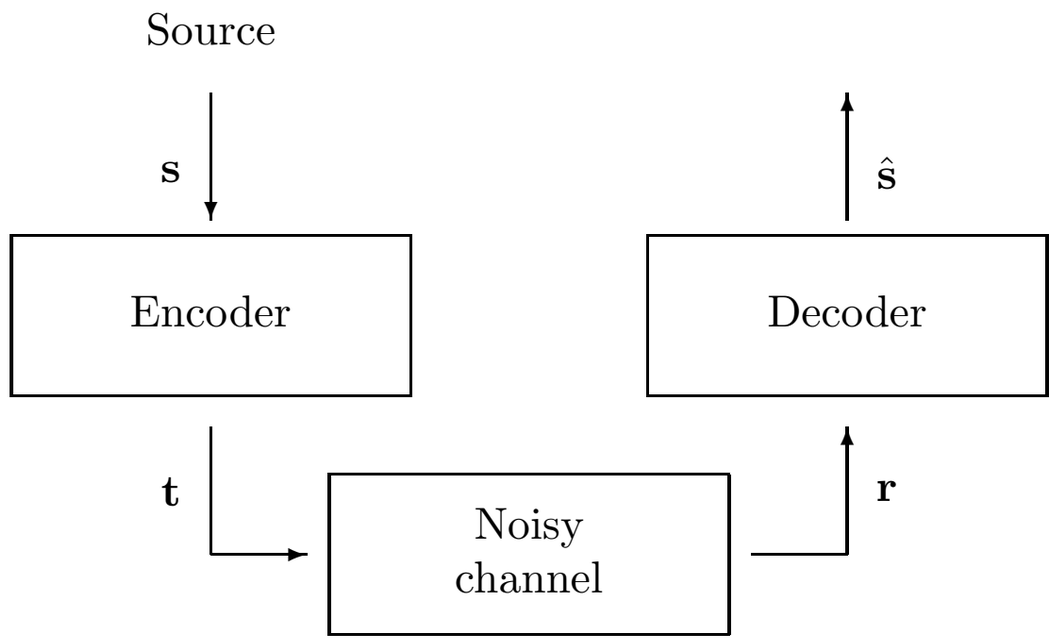
What is Information?



Information is something sent from source, and needs to be *accurately reproduced* by the receiver.
This process is called communication.

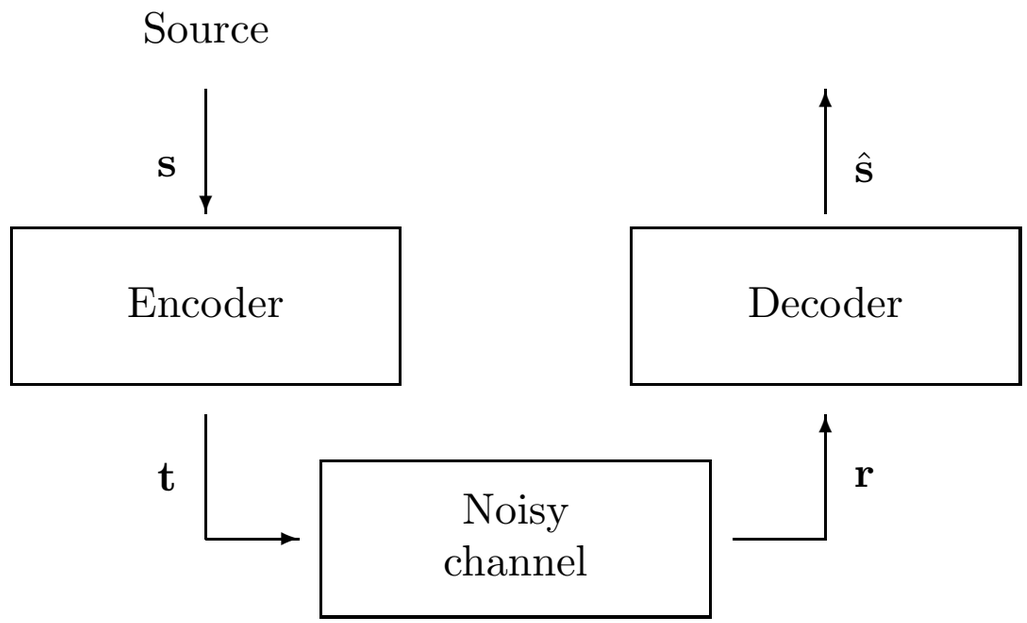
Model of Communication

- Source information set: The candidate set of information to send.
- Encoding: Turn chosen information into code.
- The code is sent over the noisy channel, possible to be corrupted due to noise.
- The decoder receives the code, and tries to recognize the original information.



Fundamental Questions

- How to encode information?
- The limit of the shortest code length for information compression?
- The limit of the fastest speed for noisy channel communication?
- Practical ways for reaching the above limits?

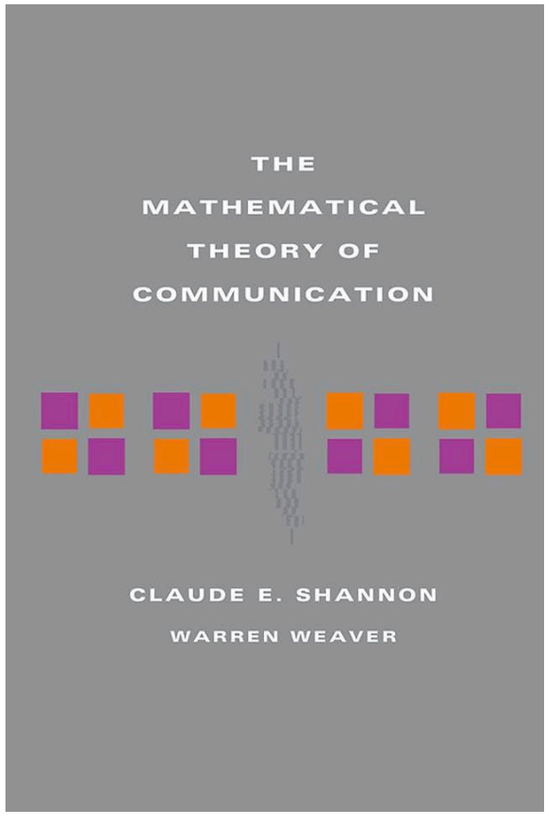


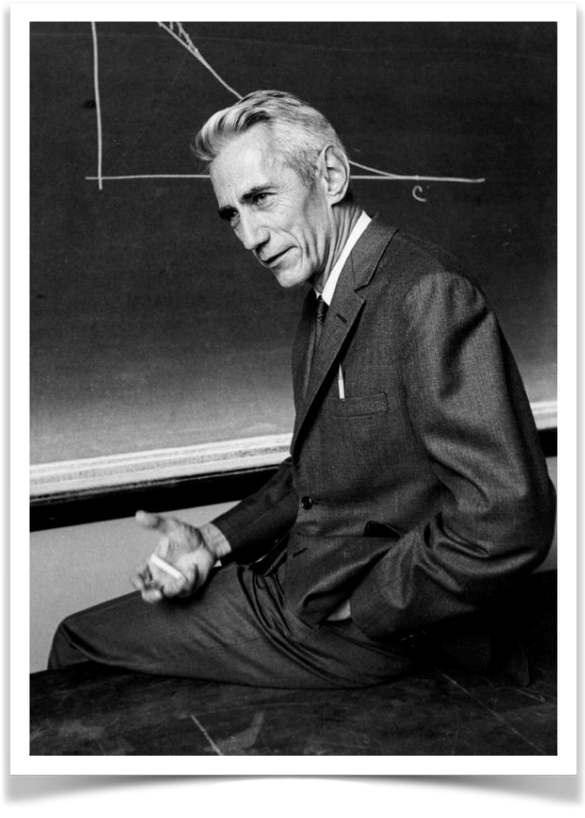
Information Theory

A Mathematical Theory of Communication

By C. E. SHANNON

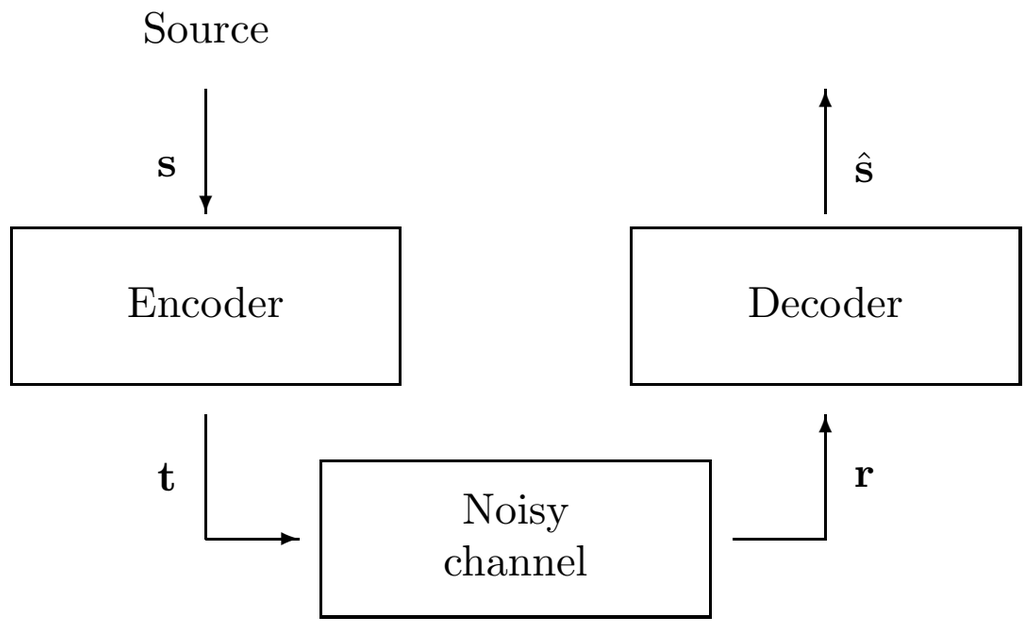
Reprinted with corrections from *The Bell System Technical Journal,*
Vol. 27, pp. 379–423, 623–656, July, October, 1948.





Fundamental Questions

- How to encode information?
- The limit of the shortest code length for information compression?
- The limit of the fastest speed for noisy channel communication?
- Practical ways for reaching the above limits?

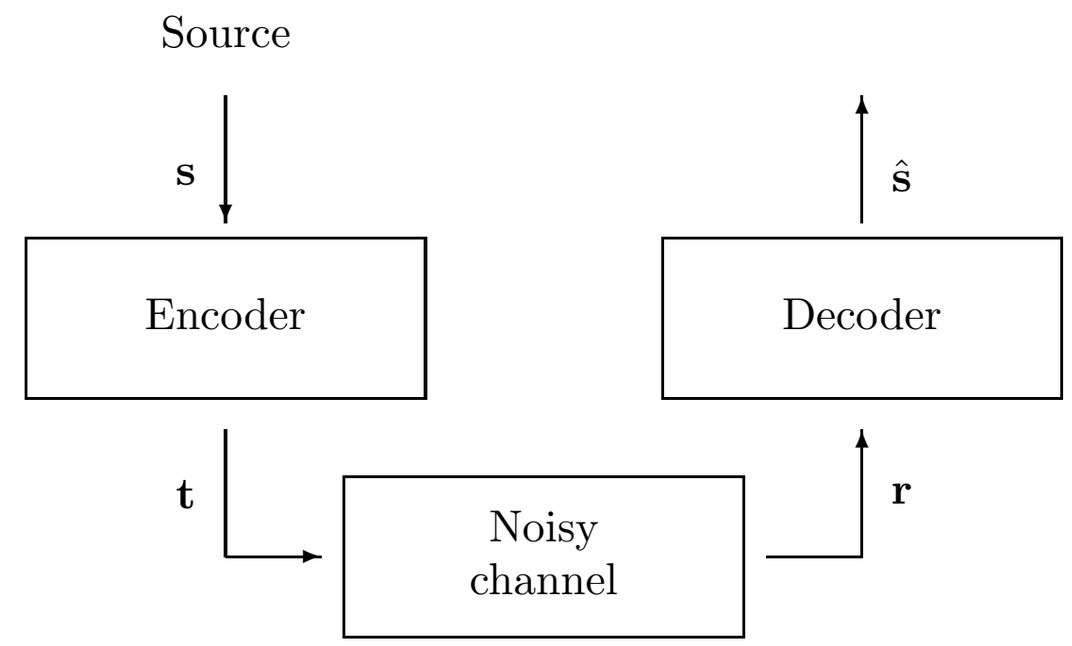


Mathematical Model of Communication

- Public information set:

Representing Information as random variable

N elements, with distribution of occurrence $P = (p_1, p_2, \ldots, p_N)$

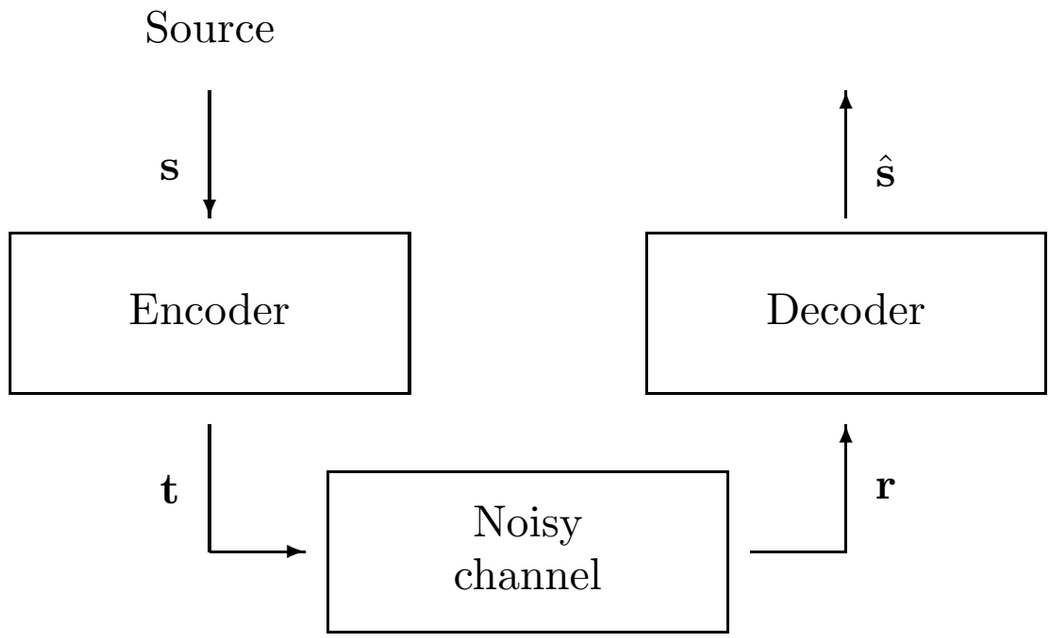


Mathematical Model of Communication

- Public information set:

  N elements, with distribution of occurrence $P = (p_1, p_2, \ldots, p_N)$

Representing Information as random variable



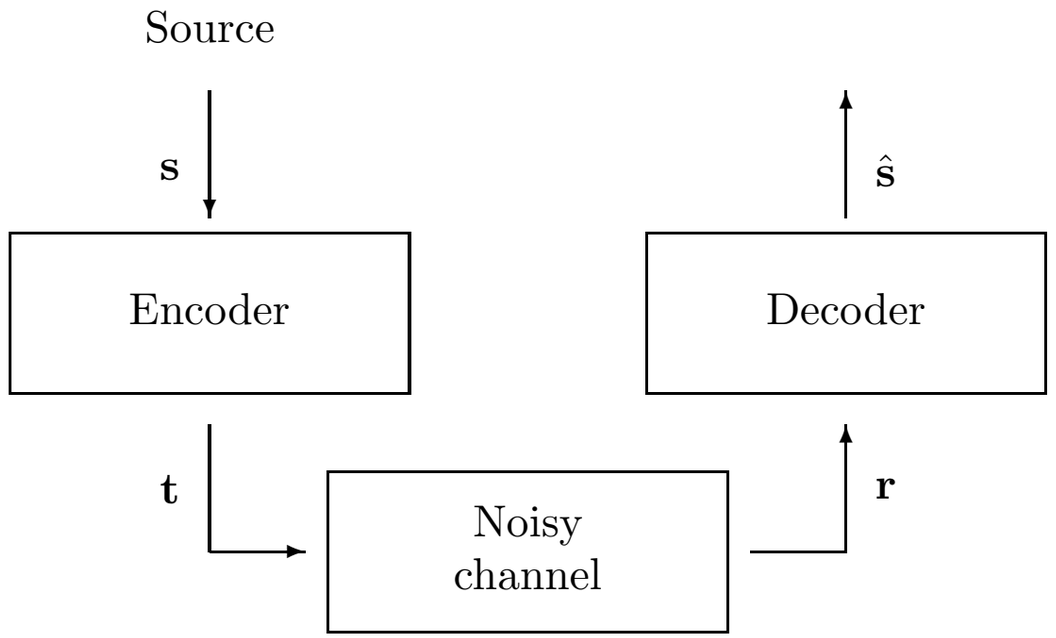
- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by a given protocol, each bin is called a bit, e.g. 10101.

Mathematical Model of Communication

- Public information set:

  Representing Information as random variable

  N elements, with distribution of occurrence $P = (p_1, p_2, \ldots, p_N)$



- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by a given protocol, each bin is called a bit, e.g. 10101.

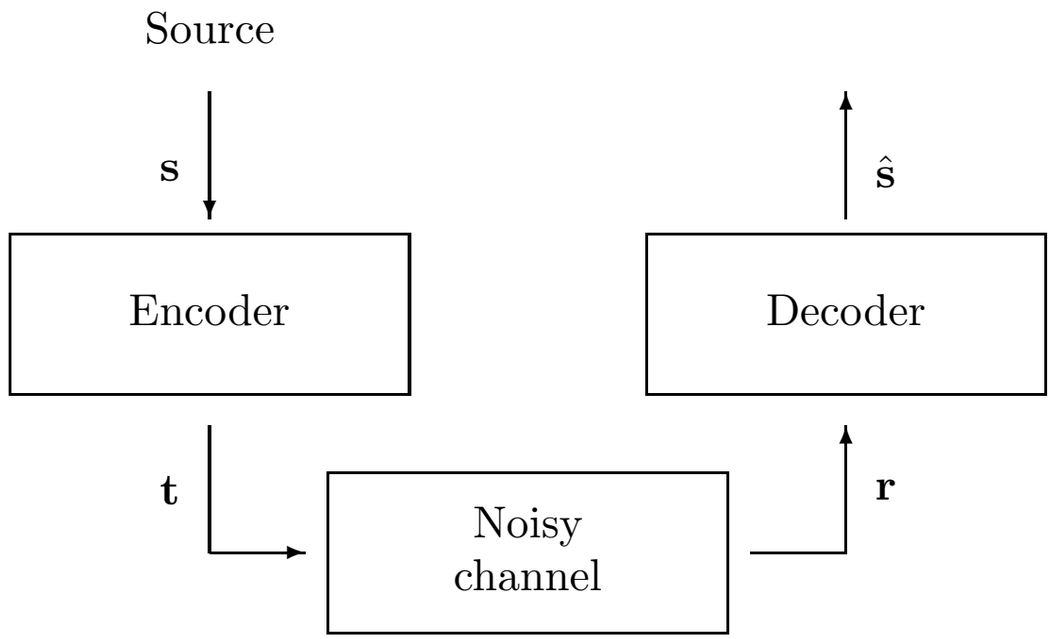
- The code is sent over the noisy channel, each bit can be flipped due to noise.

Mathematical Model of Communication

- Public information set:

  Representing Information as random variable

  N elements, with distribution of occurrence $P = (p_1, p_2, \ldots, p_N)$



- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by a given protocol, each bin is called a bit, e.g. 10101.

- The code is sent over the noisy channel, each bit can be flipped due to noise.

- The decoder receives the code, and tries to recognize the original elements by the given protocol.

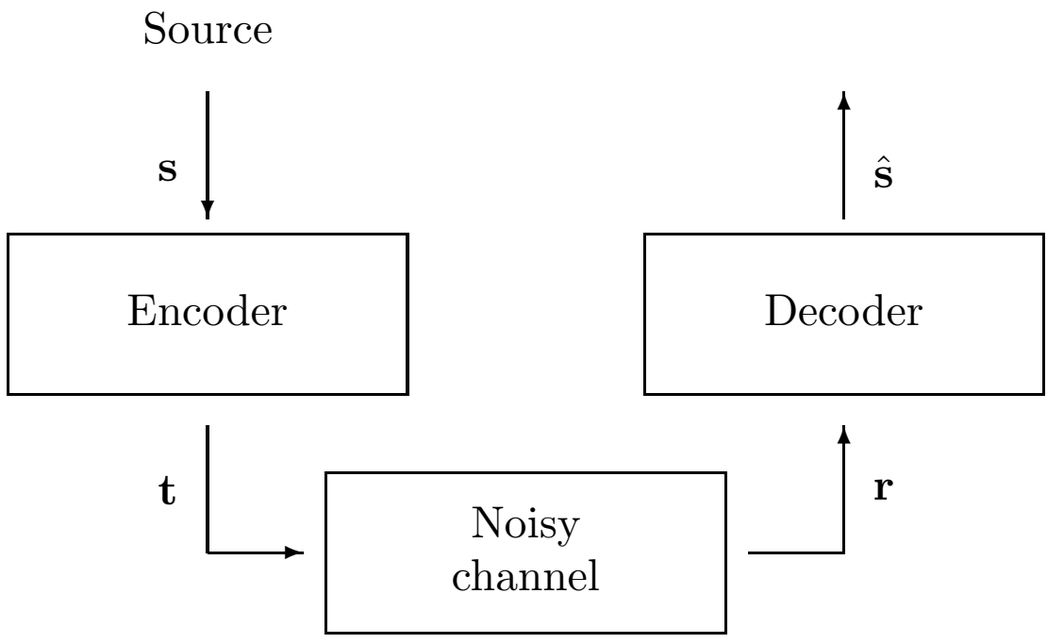
Mathematical Model of Communication

- Public information set:

Representing Information as random variable

N elements, with distribution of occurrence $P = (p_1, p_2, \ldots, p_N)$

- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by a given protocol, each bin is called a bit, e.g. 10101.

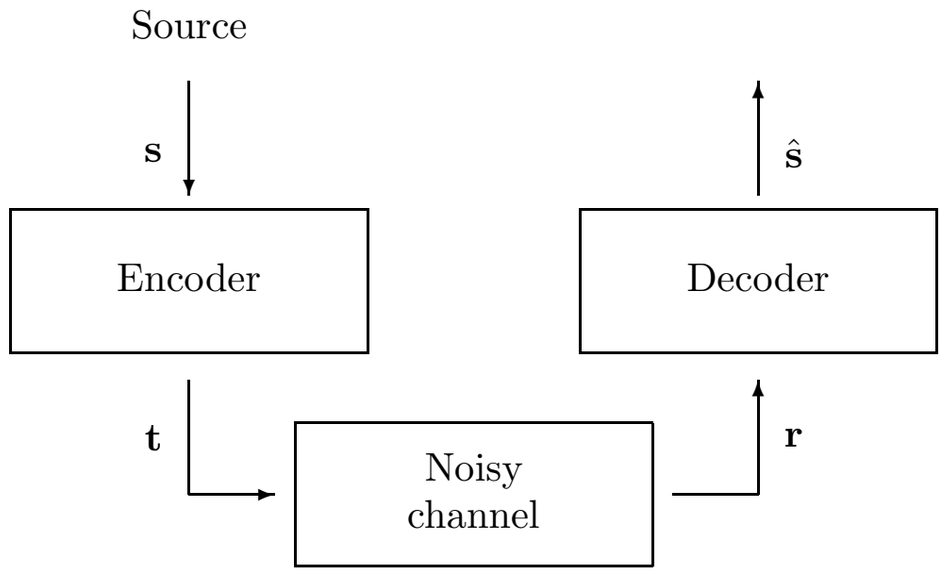


- The code is sent over the noisy channel, each bit can be flipped due to noise.

- The decoder receives the code, and tries to recognize the original elements by the given protocol.

Source and Noisy Channel Encoding

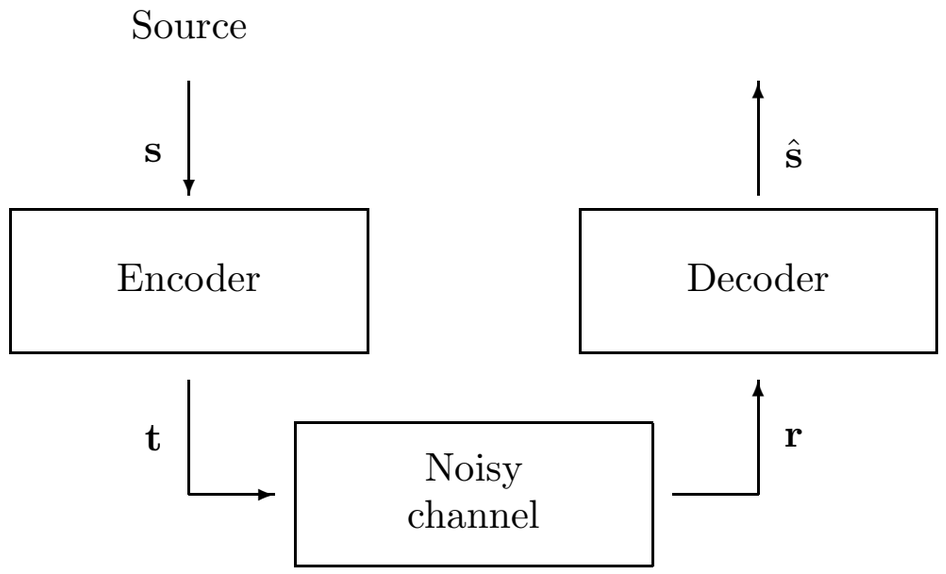
- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by source coding and noisy channel coding protocols.



Source and Noisy Channel Encoding

- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by source coding and noisy channel coding protocols.

- Source coding protocol: Represent each element as a fixed binary code. Given a sequence of elements, just concatenate the codes. The target is to make the expected code length shorter.

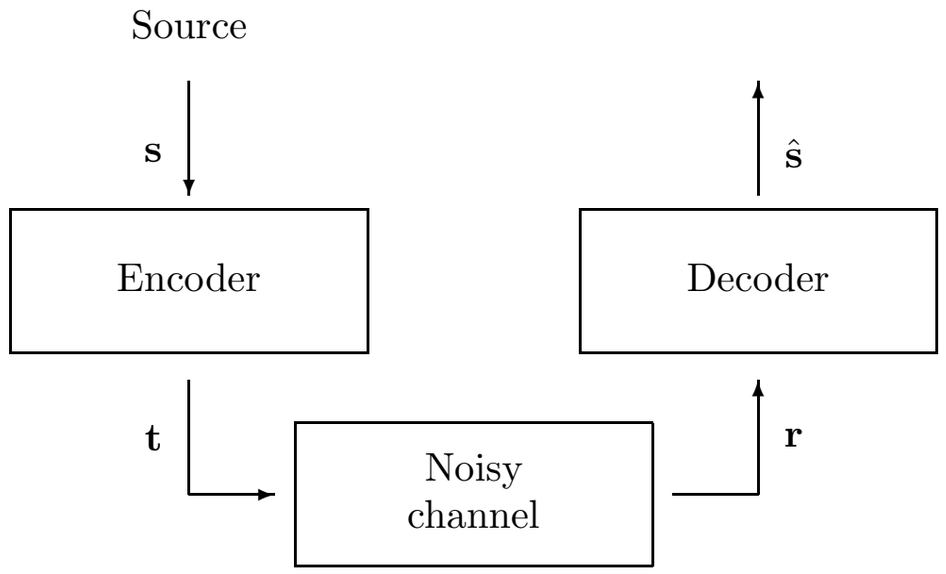


Source and Noisy Channel Encoding

- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by source coding and noisy channel coding protocols.

- Source coding protocol: Represent each element as a fixed binary code. Given a sequence of elements, just concatenate the codes. The target is to make the expected code length shorter.

- Noisy channel coding protocol: Add redundant bits in source codes to make denoising possible.

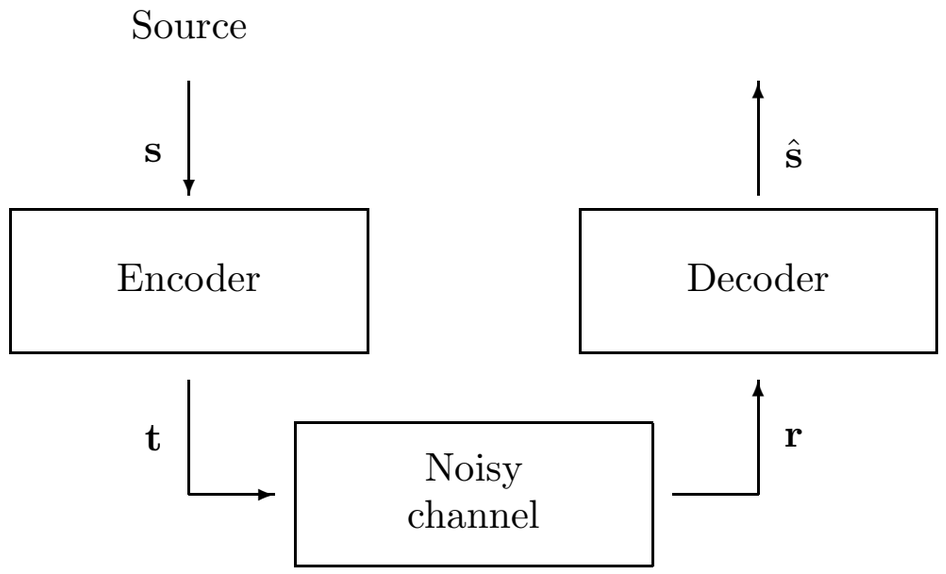


Source and Noisy Channel Encoding

- Encoding: Given a sequence of M elements sampled i.i.d from P, the encoder generate a binary code by source coding and noisy channel coding protocols.

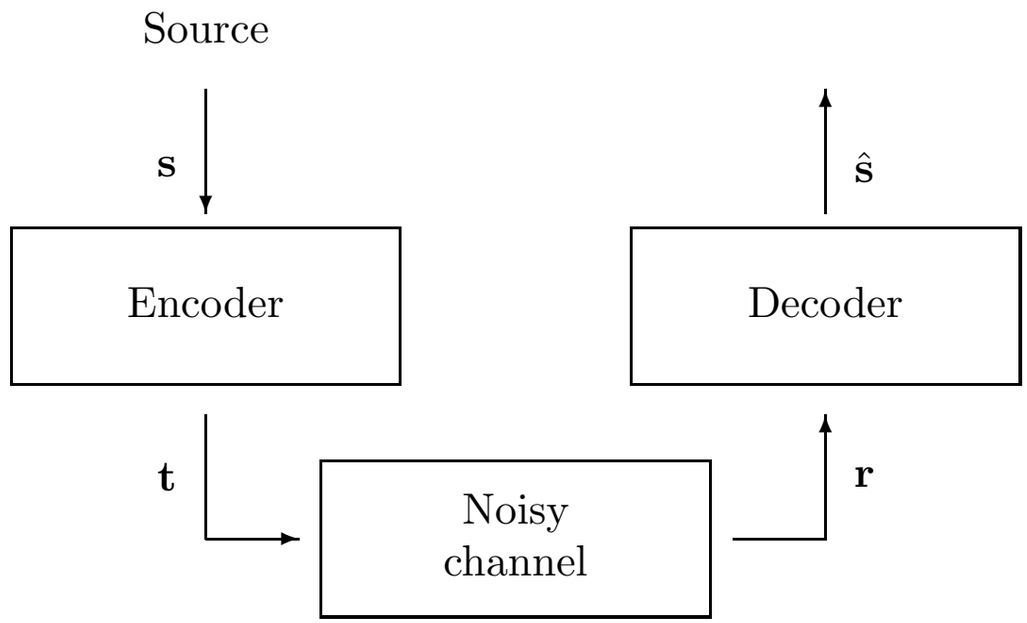
- Source coding protocol: Represent each element as a fixed binary code. Given a sequence of elements, just concatenate the codes. The target is to make the expected code length shorter.

- Noisy channel coding protocol: Add redundant bits in source codes to make denoising possible.



Fundamental Questions

- How to encode information?
- The limit of the shortest code length for information compression?
- The limit of the fastest speed for noisy channel communication?
- Practical ways for reaching the above limits?



Source Encoding

- Source coding protocol: Represent each element as a fixed binary code. Given a sequence of elements, just concatenate the codes. The target is to make the expected code length shorter.

- Consider sending the competition result of Chinese football team:

  Win: p=0.1 Draw: p=0.3 Lose: p=0.6

- Prefix-free coding:
  - No ambiguous for decoding
  - Apply shorter codes for high-probability elements can reduce the expected code length.
  - We can not make code length arbitrarily short: For prefix-free codes, Kraft's inequality holds:

Win	Draw	Lose
11	10	0

$$\sum_{i=1}^{N} 2^{-L_i} \leq 1$$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$
- The entropy of a random variable (or a prob. distribution) is the expected information contained:

$$H(x) = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}$$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$
- The entropy of a random variable (or a prob. distribution) is the expected information contained:

$$H(x) = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}$$

Source coding theorem:
Entropy is the (exact) lower bound for source coding length!

Asymptotic Equipartition Principle

- Why entropy is called entropy? The AEP property:

'Asymptotic equipartition' principle. For an ensemble of N independent identically distributed (i.i.d.) random variables $X^N \equiv (X_1, X_2, \ldots, X_N)$, with N sufficiently large, the outcome $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ is almost certain to belong to a subset of $\mathcal{A}_X^N$ having only $2^{NH(X)}$ members, each having probability 'close to' $2^{-NH(X)}$.

Notice that if $H(X) < H_0(X)$ then $2^{NH(X)}$ is a *tiny* fraction of the number of possible outcomes $|\mathcal{A}_X^N| = |\mathcal{A}_X|^N = 2^{NH_0(X)}$.

A direct consequence of the law of large numbers

Outline: Greedy Algorithms

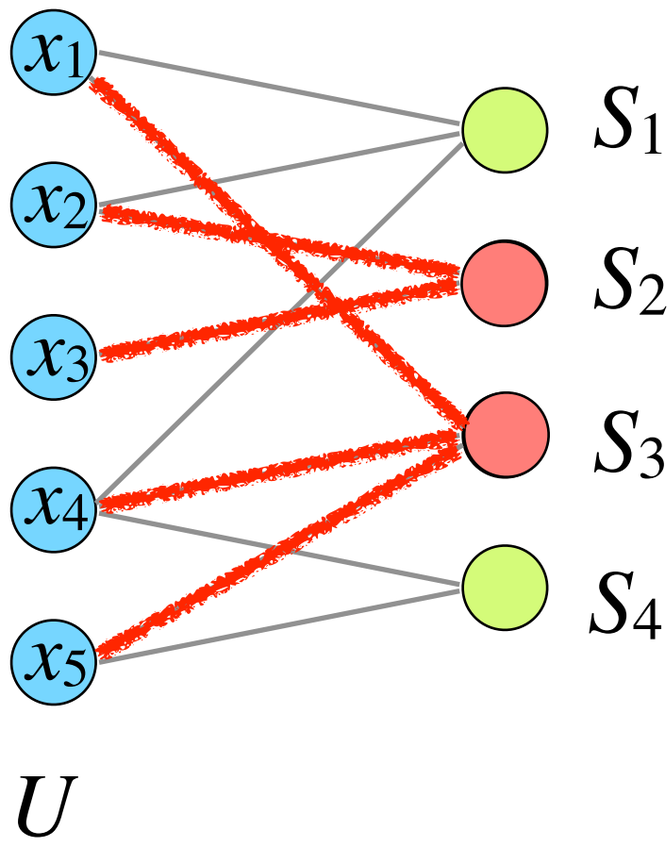
- Active selection
- Huffman coding
- Set cover
- Take-home messages

Slides courtesy: https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

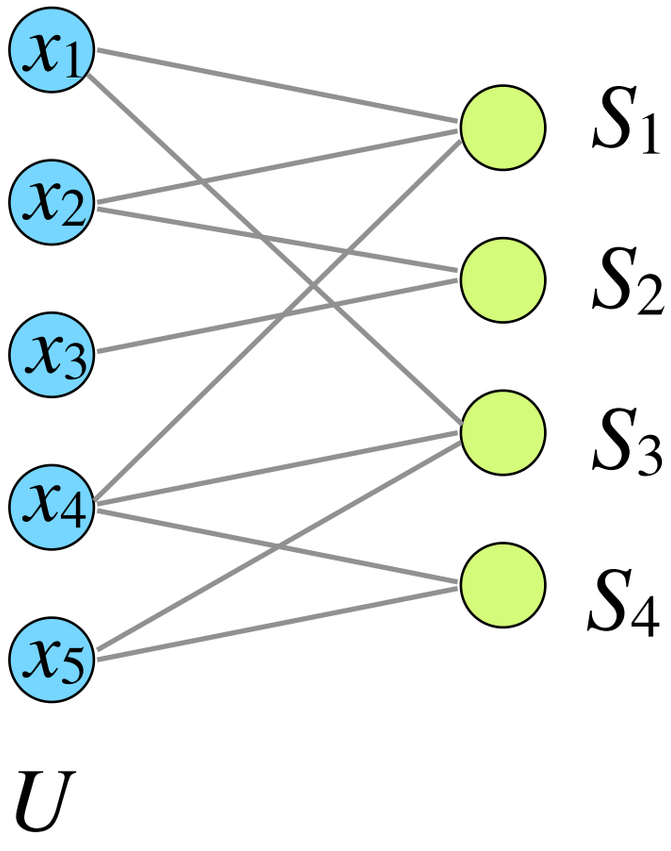
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



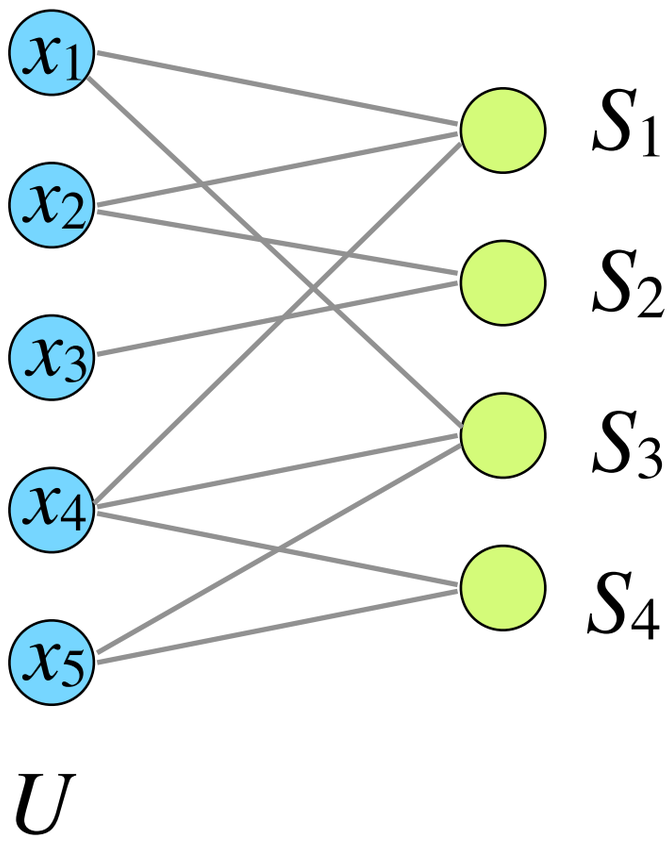
- **NP**-hard
- one of Karp's 21 **NP**-complete problems
- *frequency*: # of sets an element is in

$$frequency(x) = |\{S_i : x \in S_i\}|$$

Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

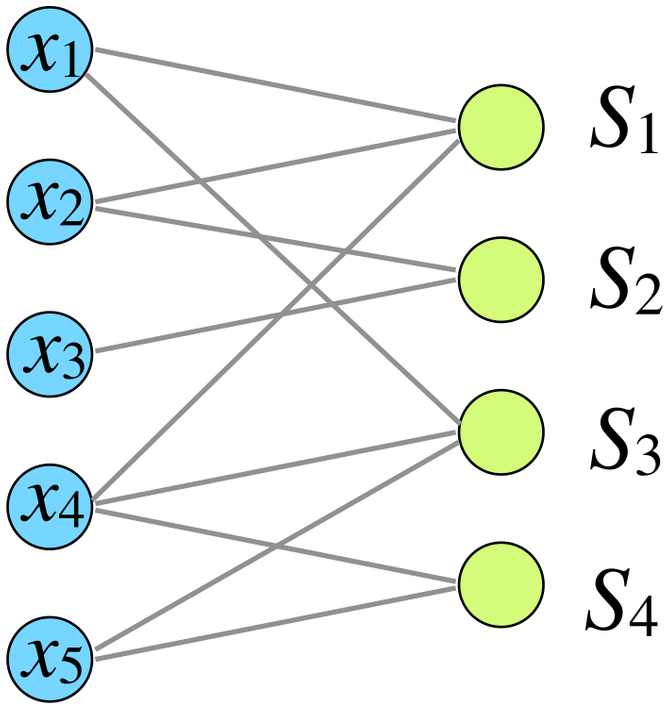
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$;

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$;

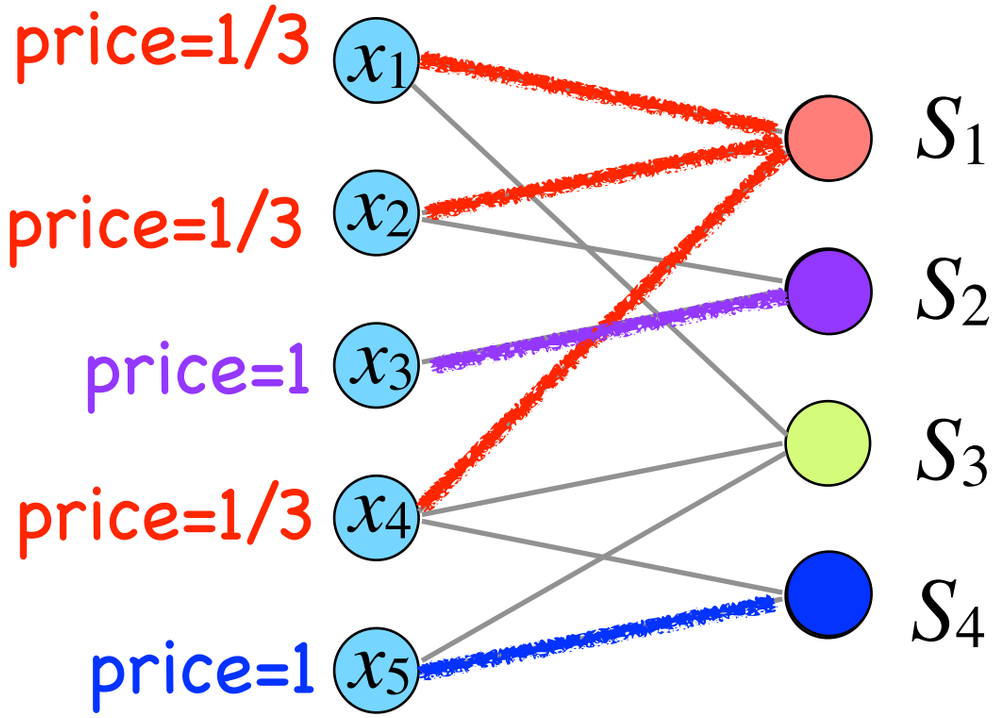
OPT(I): value of minimum set cover of instance I

SOL(I): value of the set cover returned by the **GreedyCover** algorithm on instance I

GreedyCover has *approximation ratio* α if

$$\forall \text{ instance } I, \quad \frac{\mathrm{SOL}(I)}{\mathrm{OPT}(I)} \leq \alpha$$

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; ⌜∀x∈Sᵢ∩U, price(x)=1/|Sᵢ∩U|

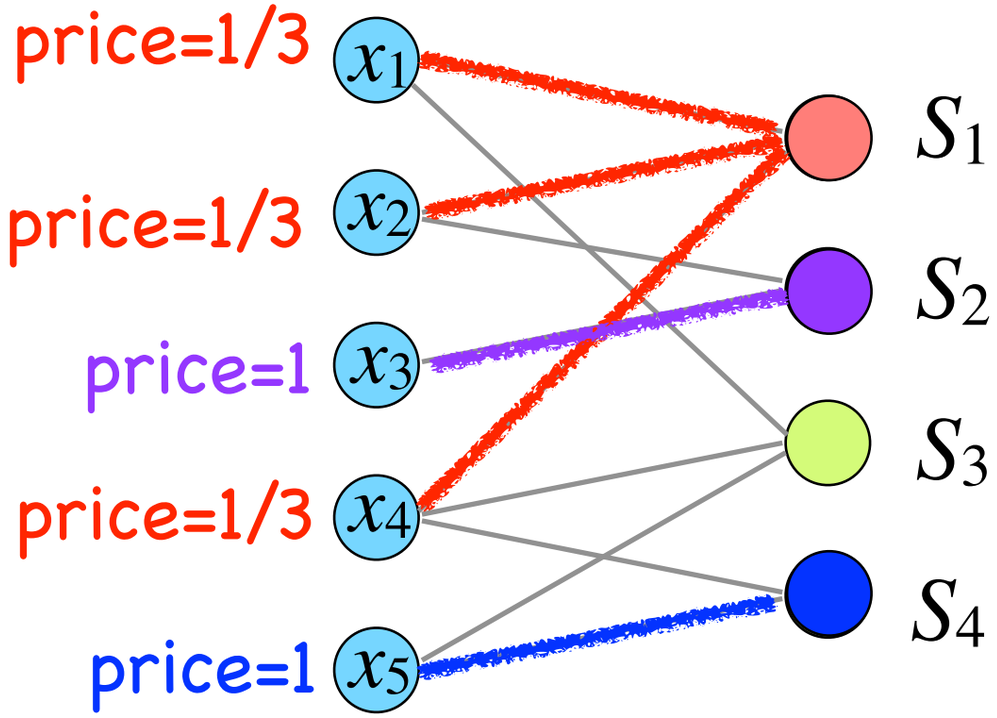
$$|C| = \sum_{x \in U} price(x)$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

elements can be *matched* to the sets in OPT cover

$$\exists S_i, \ |S_i| \geq \frac{|U|}{OPT} \Rightarrow price(x_1) \leq \frac{OPT}{|U|}$$

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; ∀x∈S_i, price(x)=1/$|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x)$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

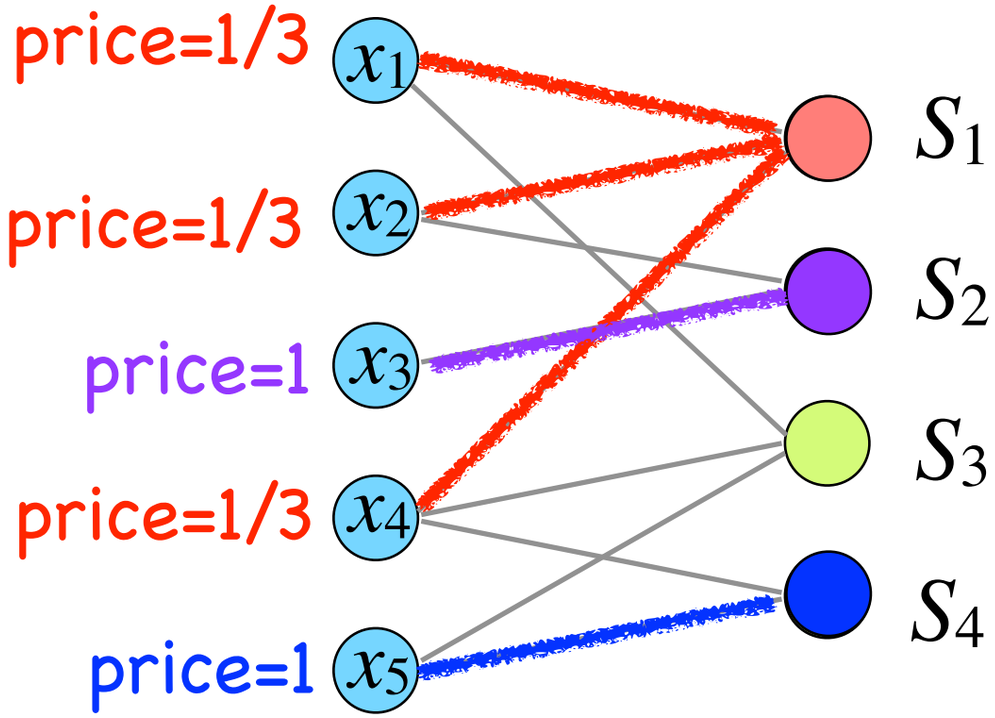
$$price(x_k) \leq \frac{OPT}{n-k+1}$$

consider U_t in iteration t where x_k is covered:

$|U_t| \geq n\text{-}k\text{+}1$

all $S_i \cap U_t$ form a set cover instance: ≤OPT

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; $\forall x \in S_i$, $price(x) = 1/|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = H_n \cdot OPT$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

$$price(x_k) \leq \frac{OPT}{n-k+1}$$

GreedyCover

Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add i with largest $|S_i\cap U|$ to C;
  $U = U \setminus S_i$;

- *GreedyCover* has approximation ratio $H_n \approx \ln n + O(1)$.
- [Lund, Yannakakis 1994; Feige 1998] There is no poly-time (1-o(1))ln n-approx. algorithm unless **NP** = quasi-poly-time.
- [Ras, Safra 1997] For some c there is no poly-time c ln n-approximation algorithm unless **NP = P**.
- [Dinur, Steuer 2014] There is no poly-time (1-o(1))ln n-approximation algorithm unless **NP = P**.

Outline: Greedy Algorithms

- Active selection
- Huffman coding
- Set cover
- Take-home messages

Take-Home Messages

- Greedy algorithms solves optimization problems with specific sub-problem structures in comparison with dynamic programming.

- Even when the algorithm is actually sub-optimal, sometimes can lead to good approximation algorithms.

Take-Home Messages

- Greedy algorithms solves optimization problems with specific sub-problem structures in comparison with dynamic programming.

- Even when the algorithm is actually sub-optimal, sometimes can lead to good approximation algorithms.

Try to do more exercises.

Thanks for your attention!
Discussions?

Reference

Algorithms (DPV): Chap. 5.

Introduction to Algorithms (4th Edition): Chap.15.

https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

Book of Information Theory: https://www.inference.org.uk/itprnn/book.pdf